SQLite Tutorial



SQLITE TUTORIAL

Simply Easy Learning by tutorialspoint.com

tutorialspoint.com

ABOUT THE TUTORIAL

SQLite Tutorial

SQLite is a software library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine. SQLite is the most widely deployed SQL database engine in the world. The source code for SQLite is in the public domain.

This tutorial will give you quick start with SQLite and make you comfortable with SQLite programming.

Audience

This reference has been prepared for the beginners to help them understand the basic to advanced concepts related to SQLite Database Engine.

Prerequisites

Before you start doing practice with various types of examples given in this reference, I'm making an assumption that you are already aware about what is database, especially RDBMS and what is a computer programming language.

Copyright & Disclaimer Notice

Table of Content

CHAPTER

1

SQLite Overview

This tutorial helps you to understand what is SQLite , how it differs from SQL, why it is needed and the way in which it handles the applications Database.

SQLite is a software library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine. SQLite is one of the fastest-growing database engines around, but that's growth in terms of popularity, not anything to do with its size. The source code for SQLite is in the public domain.

What is SQLite?

SQLite is an in-process library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine. It is the one database, which is zero-configured, that means like other database you do not need to configure it in your system.

SQLite engine is not a standalone process like other databases, you can link it statically or dynamically as per your requirement with your application. The SQLite accesses its storage files directly.

Why SQLite?

- SQLite does not require a separate server process or system to operate (serverless).
- SQLite comes with zero-configuration, which means no setup or administration needed.
- A complete SQLite database is stored in a single cross-platform disk file.
- SQLite is very small and light weight, less than 400KiB fully configured or less than 250KiB with optional features omitted.
- SQLite is self-contained, which means no external dependencies.
- SQLite transactions are fully ACID-compliant, allowing safe access from multiple processes or threads.
- SQLite supports most of the query language features found in the SQL92 (SQL2) standard.
- SQLite is written in ANSI-C and provides simple and easy-to-use API.
- SQLite is available on UNIX (Linux, Mac OS-X, Android, iOS) and Windows (Win32, WinCE, WinRT).

History:

1. 2000 -- D. Richard Hipp had designed SQLite for the purpose of no administration required for operating a program.

2. 2000 -- In August, SQLite 1.0 released with GNU Database Manager.

3. 2011 -- Hipp announced to add UNQl interface to SQLite DB and to develop UNQLite (Document oriented database).

SQLite Limitations:

There are few unsupported features of SQL92 in SQLite which are shown below:

Feature	Description
RIGHT OUTER JOIN	Only LEFT OUTER JOIN is implemented.
FULL OUTER JOIN	Only LEFT OUTER JOIN is implemented.
ALTER TABLE	The RENAME TABLE and ADD COLUMN variants of the ALTER TABLE command are supported. The DROP COLUMN, ALTER COLUMN, ADD CONSTRAINT not supported.
Trigger support	FOR EACH ROW triggers are supported but not FOR EACH STATEMENT triggers.
VIEWs	VIEWs in SQLite are read-only. You may not execute a DELETE, INSERT, or UPDATE statement on a view.
GRANT and REVOKE	The only access permissions that can be applied are the normal file access permissions of the underlying operating system.

SQLite Commands:

The standard SQLite commands to interact with relational databases are similar as SQL. They are CREATE, SELECT, INSERT, UPDATE, DELETE and DROP. These commands can be classified into groups based on their operational nature:

DDL - Data Definition Language:

Command	Description
CREATE	Creates a new table, a view of a table, or other object in database
ALTER	Modifies an existing database object, such as a table.
DROP	Deletes an entire table, a view of a table or other object in the database.

DML - Data Manipulation Language:

Command	Description
INSERT	Creates a record
UPDATE	Modifies records

DELETE	Deletes records

DQL - Data Query Language:

Command	Description
SELECT	Retrieves certain records from one or more tables

CHAPTER

2

SQLite Installation

The SQLite is famous for its great feature zero-configuration, which means no complex setup or administration is needed. This chapter will take you through the process of setting up SQLite on Windows, Linux and Mac OS X.

Install SQLite On Windows

- Go to SQLite download page, and download precompiled binaries from Windows section.
- You will need to download **sqlite-shell-win32-*.zip** and **sqlite-dll-win32-*.zip** zipped files.
- Create a folder C:\>sqlite and unzip above two zipped files in this folder which will give you sqlite3.def, sqlite3.dll and sqlite3.exe files.
- Add C:\>sqlite in your PATH environment variable and finally go to the command prompt and issue **sqlite3** command, which should display a result something as below.

```
C:\>sqlite3
SQLite version 3.7.15.2 2013-01-09 11:53:05
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite>
```

Install SQLite On Linux

Today, almost all the flavours of Linux OS are being shipped with SQLite. So you just issue the following command to check if you already have SQLite installed on your machine or not.

```
$sqlite3
SQLite version 3.7.15.2 2013-01-09 11:53:05
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite>
```

If you do not see above result, then it means you do not have SQLite installed on your Linux machine. So let's follow the following steps to install SQLite:

- Go to SQLite download page and download **sqlite-autoconf-*.tar.gz** from source code section.
- Follow the following steps:

```
$tar xvfz sqlite-autoconf-3071502.tar.gz
$cd sqlite-autoconf-3071502
$./configure --prefix=/usr/local
$make
$make install
```

Above procedure will end with SQLite installation on your Linux machine which you can verify as explained above.

Install SQLite On Mac OS X

Though latest version of Mac OS X comes pre-installed with SQLite but if you do not have installation available then just follow the following steps:

- Go to SQLite download page and download **sqlite-autoconf-*.tar.gz** from source code section.
- Follow the following steps:

```
$tar xvfz sqlite-autoconf-3071502.tar.gz
$cd sqlite-autoconf-3071502
$./configure --prefix=/usr/local
$make
$make install
```

Above procedure will end with SQLite installation on your Mac OS X machine which you can verify by issuing following command:

```
$sqlite3
SQLite version 3.7.15.2 2013-01-09 11:53:05
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite>
```

Finally, you have SQLite command prompt where you can issue SQLite commands to do your excercises.

CHAPTER

3

SQLite Commands

This chapter will take you through simple and useful commands used by SQLite programmers. These commands are called SQLite dot commands and exception with these commands is that they should not be terminated by a semi-colon (;).

Let's start with typing a simple **sqlite3** command at command prompt which will provide you SQLite command prompt where you will issue various SQLite commands.

```
$sqlite3
SQLite version 3.3.6
Enter ".help" for instructions
sqlite>
```

For a listing of the available dot commands, you can enter ".help" at any time. For example:

```
sqlite>.help
```

Above command will display a list of various important SQLite dot commands, which are as follows:

Command	Description
.backup ?DB? FILE	Backup DB (default "main") to FILE
.bail ON|OFF	Stop after hitting an error. Default OFF
.databases	List names and files of attached databases
.dump ?TABLE?	Dump the database in an SQL text format. If TABLE specified, only dump tables matching LIKE pattern TABLE.
.echo ON|OFF	Turn command echo on or off
.exit	Exit SQLite prompt
.explain ON|OFF	Turn output mode suitable for EXPLAIN on or off. With no args, it turns EXPLAIN on.
.header(s) ON|OFF	Turn display of headers on or off
.help	Show this message
.import FILE TABLE	Import data from FILE into TABLE
.indices ?TABLE?	Show names of all indices. If TABLE specified, only show indices for tables matching

	LIKE pattern TABLE.
.load FILE ?ENTRY?	Load an extension library
.log FILE|off	Turn logging on or off. FILE can be stderr/stdout
.mode MODE	Set output mode where MODE is one of: **csv** Comma-separated values **column** Left-aligned columns. **html** HTML <table> code **insert** SQL insert statements for TABLE **line** One value per line **list** Values delimited by .separator string **tabs** Tab-separated values **tcl** TCL list elements
.nullvalue STRING	Print STRING in place of NULL values
.output FILENAME	Send output to FILENAME
.output stdout	Send output to the screen
.print STRING...	Print literal STRING
.prompt MAIN CONTINUE	Replace the standard prompts
.quit	Exit SQLite prompt
.read FILENAME	Execute SQL in FILENAME
.schema ?TABLE?	Show the CREATE statements. If TABLE specified, only show tables matching LIKE pattern TABLE.
.separator STRING	Change separator used by output mode and .import
.show	Show the current values for various settings
.stats ON|OFF	Turn stats on or off
.tables ?PATTERN?	List names of tables matching a LIKE pattern
.timeout MS	Try opening locked tables for MS milliseconds
.width NUM NUM	Set column widths for "column" mode
.timer ON|OFF	Turn the CPU timer measurement on or off

Let's try **.show** command to see default setting for your SQLite command prompt.

```
sqlite>.show
     echo: off
  explain: off
  headers: off
     mode: column
nullvalue: ""
   output: stdout
separator: "|"
    width:
sqlite>
```

Make sure there is no space in between sqlite> prompt and dot command, otherwise it will not work.

Formatting output

You can use the following sequence of dot commands to format your output the way I have listed down in this tutorial:

```
sqlite>.header on
sqlite>.mode column
sqlite>.timer on
sqlite>
```

Above setting will produce the output in the following format:

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
CPU Time: user 0.000000 sys 0.000000
```

The sqlite_master Table

The master table holds the key information about your database tables and it is called **sqlite_master**. You can see its schema as follows:

```
sqlite>.schema sqlite_master
```

This will produce the following result:

```
CREATE TABLE sqlite_master (
  type text,
  name text,
  tbl_name text,
  rootpage integer,
  sql text
);
```

CHAPTER 4

SQLite Syntax

SQLite is followed by unique set of rules and guidelines called Syntax. This tutorial gives you a quick start with SQLite by listing all the basic SQLite Syntax.

Case Sensitivity

Important point to be noted is that SQLite is **case insensitive,** but there are some commands, which are case sensitive like **GLOB** and **glob** have different meaning in SQLite statements.

Comments

SQLite comments are extra notes, which you can add in your SQLite code to increase its readability and they can appear anywhere; whitespace can occur, including inside expressions and in the middle of other SQL statements but they can not be nested.

SQL comments begin with two consecutive "-" characters (ASCII 0x2d) and extend up to and including the next newline character (ASCII 0x0a) or until the end of input, whichever comes first.

You can also use C-style comments, which begin with "/*" and extend up to and including the next "*/" character pair or until the end of input, whichever comes first. C-style comments can span multiple lines.

```
sqlite>.help -- This is a single line comment
```

SQLite Statements

All the SQLite statements start with any of the keywords like SELECT, INSERT, UPDATE, DELETE, ALTER, DROP, etc., and all the statements end with a semicolon (;).

SQLite ANALYZE Statement:

```
ANALYZE;
or
ANALYZE database_name;
or
ANALYZE database_name.table_name;
```

SQLite AND/OR Clause:

```
SELECT column1, column2....columnN
FROM   table_name
WHERE  CONDITION-1 {AND|OR} CONDITION-2;
```

SQLite ALTER TABLE Statement:

```
ALTER TABLE table_name ADD COLUMN column_def...;
```

SQLite ALTER TABLE Statement (Rename):

```
ALTER TABLE table_name RENAME TO new_table_name;
```

SQLite ATTACH DATABASE Statement:

```
ATTACH DATABASE 'DatabaseName' As 'Alias-Name';
```

SQLite BEGIN TRANSACTION Statement:

```
BEGIN;
or
BEGIN EXCLUSIVE TRANSACTION;
```

SQLite BETWEEN Clause:

```
SELECT column1, column2....columnN
FROM   table_name
WHERE  column_name BETWEEN val-1 AND val-2;
```

SQLite COMMIT Statement:

```
COMMIT;
```

SQLite CREATE INDEX Statement:

```
CREATE INDEX index_name
ON table_name ( column_name COLLATE NOCASE );
```

SQLite CREATE UNIQUE INDEX Statement:

```
CREATE UNIQUE INDEX index_name
ON table_name ( column1, column2,...columnN);
```

SQLite CREATE TABLE Statement:

```
CREATE TABLE table_name(
   column1 datatype,
   column2 datatype,
   column3 datatype,
   .....
   columnN datatype,
   PRIMARY KEY( one or more columns )
);
```

SQLite CREATE TRIGGER Statement :

```
CREATE TRIGGER database_name.trigger_name
BEFORE INSERT ON table_name FOR EACH ROW
BEGIN
   stmt1;
   stmt2;
   ....
END;
```

SQLite CREATE VIEW Statement :

```
CREATE VIEW database_name.view_name  AS
SELECT statement....;
```

SQLite CREATE VIRTUAL TABLE Statement:

```
CREATE VIRTUAL TABLE database_name.table_name USING weblog( access.log );
or
CREATE VIRTUAL TABLE database_name.table_name USING fts3( );
```

SQLite COMMIT TRANSACTION Statement:

```
COMMIT;
```

SQLite COUNT Clause:

```
SELECT COUNT(column_name)
FROM   table_name
WHERE  CONDITION;
```

SQLite DELETE Statement:

```
DELETE FROM table_name
WHERE  {CONDITION};
```

SQLite DETACH DATABASE Statement:

```
DETACH DATABASE 'Alias-Name';
```

SQLite DISTINCT Clause:

```
SELECT DISTINCT column1, column2....columnN
FROM   table_name;
```

SQLite DROP INDEX Statement :

```
DROP INDEX database_name.index_name;
```

SQLite DROP TABLE Statement:

```
DROP TABLE database_name.table_name;
```

SQLite DROP VIEW Statement :

```
DROP INDEX database_name.view_name;
```

SQLite DROP TRIGGER Statement :

```
DROP INDEX database_name.trigger_name;
```

SQLite EXISTS Clause:

```
SELECT column1, column2....columnN
FROM   table_name
WHERE  column_name EXISTS (SELECT * FROM   table_name );
```

SQLite EXPLAIN Statement :

```
EXPLAIN INSERT statement...;
or
EXPLAIN QUERY PLAN SELECT statement...;
```

SQLite GLOB Clause:

```
SELECT column1, column2....columnN
FROM   table_name
WHERE  column_name GLOB { PATTERN };
```

SQLite GROUP BY Clause:

```
SELECT SUM(column_name)
```

```
FROM   table_name
WHERE  CONDITION
GROUP BY column_name;
```

SQLite HAVING Clause:

```
SELECT SUM(column_name)
FROM   table_name
WHERE  CONDITION
GROUP BY column_name
HAVING (arithematic function condition);
```

SQLite INSERT INTO Statement:

```
INSERT INTO table_name( column1, column2....columnN)
VALUES ( value1, value2....valueN);
```

SQLite IN Clause:

```
SELECT column1, column2....columnN
FROM   table_name
WHERE  column_name IN (val-1, val-2,...val-N);
```

SQLite Like Clause:

```
SELECT column1, column2....columnN
FROM   table_name
WHERE  column_name LIKE { PATTERN };
```

SQLite NOT IN Clause:

```
SELECT column1, column2....columnN
FROM   table_name
WHERE  column_name NOT IN (val-1, val-2,...val-N);
```

SQLite ORDER BY Clause:

```
SELECT column1, column2....columnN
FROM   table_name
WHERE  CONDITION
ORDER BY column_name {ASC|DESC};
```

SQLite PRAGMA Statement:

```
PRAGMA pragma_name;

For example:

PRAGMA page_size;
PRAGMA cache_size = 1024;
PRAGMA table_info(table_name);
```

SQLite RELEASE SAVEPOINT Statement:

```
RELEASE savepoint_name;
```

SQLite REINDEX Statement:

```
REINDEX collation_name;
REINDEX database_name.index_name;
REINDEX database_name.table_name;
```

SQLite ROLLBACK Statement:

```
ROLLBACK;
or
ROLLBACK TO SAVEPOINT savepoint_name;
```

SQLite SAVEPOINT Statement:

```
SAVEPOINT savepoint_name;
```

SQLite SELECT Statement:

```
SELECT column1, column2....columnN
FROM   table_name;
```

SQLite UPDATE Statement:

```
UPDATE table_name
SET column1 = value1, column2 = value2....columnN=valueN
[ WHERE  CONDITION ];
```

SQLite VACUUM Statement:

```
VACUUM;
```

SQLite WHERE Clause:

```
SELECT column1, column2....columnN
FROM   table_name
WHERE  CONDITION;
```

CHAPTER

5

SQLite Data Type

SQLite data type is an attribute that specifies type of data of any object. Each column, variable and expression has related data type in SQLite.

You would use these data types while creating your tables. SQLite uses a more general dynamic type system. In SQLite, the datatype of a value is associated with the value itself, not with its container.

SQLite Storage Classes:

Each value stored in an SQLite database has one of the following storage classes:

Storage Class	Description
NULL	The value is a NULL value.
INTEGER	The value is a signed integer, stored in 1, 2, 3, 4, 6, or 8 bytes depending on the magnitude of the value.
REAL	The value is a floating point value, stored as an 8-byte IEEE floating point number.
TEXT	The value is a text string, stored using the database encoding (UTF-8, UTF-16BE or UTF-16LE)
BLOB	The value is a blob of data, stored exactly as it was input.

SQLite storage class is slightly more general than a datatype. The INTEGER storage class, for example, includes 6 different integer datatypes of different lengths.

SQLite Affinity Type:

SQLite supports the concept of type affinity on columns. Any column can still store any type of data but the preferred storage class for a column is called its **affinity**. Each table column in an SQLite3 database is assigned one of the following type affinities:

Affinity	Description
TEXT	This column stores all data using storage classes NULL, TEXT or BLOB.
NUMERIC	This column may contain values using all five storage classes.
INTEGER	Behaves the same as a column with NUMERIC affinity with an exception in a CAST

	expression.
REAL	Behaves like a column with NUMERIC affinity except that it forces integer values into floating point representation
NONE	A column with affinity NONE does not prefer one storage class over another and no attempt is made to coerce data from one storage class into another.

SQLite Affinity and Type Names:

Following table lists down various data type names which can be used while creating SQLite3 tables and corresponding applied affinity also has been shown:

Data Type	**Affinity**
INT INTEGER TINYINT SMALLINT MEDIUMINT BIGINT UNSIGNED BIG INT INT2 INT8	INTEGER
CHARACTER(20) VARCHAR(255) VARYING CHARACTER(255) NCHAR(55) NATIVE CHARACTER(70) NVARCHAR(100) TEXT CLOB	TEXT

BLOB no datatype specified	NONE
REAL DOUBLE DOUBLE PRECISION FLOAT	REAL
NUMERIC DECIMAL(10,5) BOOLEAN DATE DATETIME	NUMERIC

Boolean Datatype:

SQLite does not have a separate Boolean storage class. Instead, Boolean values are stored as integers 0 (false) and 1 (true).

Date and Time Datatype:

SQLite does not have a separate storage class for storing dates and/or times, but SQLite is capable of storing dates and times as TEXT, REAL or INTEGER values.

Storage Class	Date Formate
TEXT	A date in a format like "YYYY-MM-DD HH:MM:SS.SSS".
REAL	The number of days since noon in Greenwich on November 24, 4714 B.C.
INTEGER	The number of seconds since 1970-01-01 00:00:00 UTC.

You can chose to store dates and times in any of these formats and freely convert between formats using the built-in date and time functions.

CHAPTER

6

SQLite Create Database

The SQLite **sqlite3** command is used to create new SQLite database. You do not need to have any special privilege to create a database.

Syntax:

Basic syntax of sqlite3 command is as follows:

```
$sqlite3 DatabaseName.db
```

Always, database name should be unique within the RDBMS.

Example:

If you want to create new database <testDB.db>, then SQLite3 statement would be as follows:

```
$sqlite3 testDB.db
SQLite version 3.7.15.2 2013-01-09 11:53:05
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite>
```

Above command will create a file **testDB.db** in the current directory. This file will be used as database by SQLite engine. If you have noticed while creating database, sqlite3 command will provide a **sqlite>**prompt after creating database file successfully.

Once a database is created, you can check it in the list of databases using SQLite **.databases** command as follows:

```
sqlite>.databases
seq  name             file
---  ---------------  ----------------------
0    main             /home/sqlite/testDB.db
```

You will use SQLite **.quit** command to come out of the sqlite prompt as follows:

```
sqlite>.quit
$
```

The .dump Command

You can use **.dump** dot command to export complete database in a text file using SQLite command at command prompt as follows:

```
$sqlite3 testDB.db .dump > testDB.sql
```

Above command will convert the entire contents of **testDB.db** database into SQLite statements and dump it into ASCII text file **testDB.sql**. You can do restoration from the generated testDB.sql in simple way as follows:

```
$sqlite3 testDB.db < testDB.sql
```

At this moment your database is empty, so you can try above two procedures once you have few tables and data in your database. For now, let's proceed to next chapter.

CHAPTER

7

SQLite Attach Database

Consider a case when you have multiple databases available and you want to use any one of them at a time. SQLite **ATTACH DTABASE** statement is used to select a particular database, and after this command, all SQLite statements will be executed under the attached database.

Syntax:

Basic syntax of SQLite ATTACH DATABASE statement is as follows:

```
ATTACH DATABASE 'DatabaseName' As 'Alias-Name';
```

Above command will also create a database in case database is already not created, otherwise it will just attach database file name with logical database 'Alias-Name'.

Example:

If you want to attach an existing database **testDB.db**, then ATTACH DATABASE statement would be as follows:

```
sqlite> ATTACH DATABASE 'testDB.db' as 'TEST';
```

Use SQLite **.database** command to display attached database.

```
sqlite> .database
seq  name             file
---  ---------------  ----------------------
0    main             /home/sqlite/testDB.db
2    test             /home/sqlite/testDB.db
```

The database names **main** and **temp** are reserved for the primary database and database to hold temporary tables and other temporary data objects. Both of these database names exist for every database connection and should not be used for attachment, otherwise you will get a warning message something as follows:

```
sqlite>  ATTACH DATABASE 'testDB.db' as 'TEMP';
Error: database TEMP is already in use
sqlite>  ATTACH DATABASE 'testDB.db' as 'main';
Error: database TEMP is already in use
```

CHAPTER

8

SQLite Detach Database

SQLite **DETACH DTABASE** statement is used to detach and dissociate a named database from a database connection which was previously attached using ATTACH statement. If the same database file has been attached with multiple aliases, then DETACH command will disconnect only given name and rest of the attachement will still continue. You cannot detach the **main** or **temp** databases.

If the database is an in-memory or temporary database, the database will be destroyed and the contents will be lost.

Syntax:

Basic syntax of SQLite DETACH DATABASE 'Alias-Name' statement is as follows:

```
DETACH DATABASE 'Alias-Name';
```

Here 'Alias-Name' is the same alias, which you had used while attaching database using ATTACH statement.

Example:

Consider you have a database, which you created in previous chapter and attached it with 'test' and 'currentDB' as we can see using .database command:

```
sqlite>.databases
seq  name             file
---  ---------------  ----------------------
0    main             /home/sqlite/testDB.db
2    test             /home/sqlite/testDB.db
3    currentDB        /home/sqlite/testDB.db
```

Now let's try to detach 'currentDB' from testDB.db as follows:

```
sqlite> DETACH DATABASE 'currentDB';
```

Now, if you will check current attachment, you will find that testDB.db is still connected with 'test' and 'main'.

```
sqlite>.databases
seq  name             file
---  ---------------  ----------------------
0    main             /home/sqlite/testDB.db
2    test             /home/sqlite/testDB.db
```

CHAPTER

9

SQLite Create Table

The SQLite **CREATE TABLE** statement is used to create a new table in any of the given database.

Creating a basic table involves naming the table and defining its columns and each column's data type.

Syntax:

Basic syntax of CREATE TABLE statement is as follows:

```
CREATE TABLE database_name.table_name(
   column1 datatype  PRIMARY KEY(one or more columns),
   column2 datatype,
   column3 datatype,
   .....
   columnN datatype,
);
```

CREATE TABLE is the keyword telling the database system to create a new table. The unique name or identifier for the table follows the CREATE TABLE statement. Optionally you can specify database_name along with table_name.

Example:

Following is an example, which creates a COMPANY table with ID as primary key and NOT NULL are the constraints showing that these fields can not be NULL while creating records in this table:

```
sqlite> CREATE TABLE COMPANY(
   ID INT PRIMARY KEY     NOT NULL,
   NAME           TEXT    NOT NULL,
   AGE            INT     NOT NULL,
   ADDRESS        CHAR(50),
   SALARY         REAL
);
```

Let us create one more table, which we will use in our exercises in subsequent chapters:

```
sqlite> CREATE TABLE DEPARTMENT(
   ID INT PRIMARY KEY      NOT NULL,
   DEPT           CHAR(50) NOT NULL,
   EMP_ID         INT      NOT NULL
);
```

You can verify if your table has been created successfully using SQLIte command **.tables** command, which will be used to list down all the tables in an attached database.

```
sqlite>.tables
COMPANY     DEPARTMENT
```

Here, you can see COMPANY table twice because it's showing COMPANY table for main database and test.COMPANY table for 'test' alias created for your testDB.db. You can get complete information about a table using SQLite **.schema** command as follows:

```
sqlite>.schema COMPANY
CREATE TABLE COMPANY(
   ID INT PRIMARY KEY     NOT NULL,
   NAME           TEXT    NOT NULL,
   AGE            INT     NOT NULL,
   ADDRESS        CHAR(50),
   SALARY         REAL
);
```

CHAPTER

10

SQLite Drop Table

The SQLite **DROP TABLE** statement is used to remove a table definition and all associated data, indexes, triggers, constraints and permission specifications for that table.

You have to be careful while using this command because once a table is deleted then all the information available in the table would also be lost forever.

Syntax:

Basic syntax of DROP TABLE statement is as follows. You can optionally specify database name along with table name as follows:

```
DROP TABLE database_name.table_name;
```

Example:

Let us first verify COMPANY table and then we would delete it from the database.

```
sqlite>.tables
COMPANY       test.COMPANY
```

This means COMPANY table is available in the database, so let us drop it as follows:

```
sqlite>DROP TABLE COMPANY;
sqlite>
```

Now, if you would try .TABLES command, then you will not find COMPANY table anymore:

```
sqlite>.tables
sqlite>
```

It shows nothing means the table from your database has been dropped successfully.

CHAPTER

11

SQLite Insert Query

The SQLite **INSERT INTO** Statement is used to add new rows of data into a table in the database.

Syntax:

There are two basic syntaxes of INSERT INTO statement as follows:

```
INSERT INTO TABLE_NAME (column1, column2, column3,...columnN)]
VALUES (value1, value2, value3,...valueN);
```

Here, column1, column2,...columnN are the names of the columns in the table into which you want to insert data.

You may not need to specify the column(s) name in the SQLite query if you are adding values for all the columns of the table. But make sure the order of the values is in the same order as the columns in the table. The SQLite INSERT INTO syntax would be as follows:

```
INSERT INTO TABLE_NAME VALUES (value1,value2,value3,...valueN);
```

Example:

Consider you already have created COMPANY table in your testDB.db as follows:

```
sqlite> CREATE TABLE COMPANY(
   ID INT PRIMARY KEY     NOT NULL,
   NAME           TEXT    NOT NULL,
   AGE            INT     NOT NULL,
   ADDRESS        CHAR(50),
   SALARY         REAL
);
```

Now, following statements would create six records in COMPANY table:

```
INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (1, 'Paul', 32, 'California', 20000.00 );

INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (2, 'Allen', 25, 'Texas', 15000.00 );

INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (3, 'Teddy', 23, 'Norway', 20000.00 );
```

```
INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (4, 'Mark', 25, 'Rich-Mond ', 65000.00 );

INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (5, 'David', 27, 'Texas', 85000.00 );

INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (6, 'Kim', 22, 'South-Hall', 45000.00 );
```

You can create a record in COMPANY table using second syntax as follows:

```
INSERT INTO COMPANY VALUES (7, 'James', 24, 'Houston', 10000.00 );
```

All the above statements would create following records in COMPANY table. Next chapter will teach you how to display all these records from a table.

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

Populate one table using another table:

You can populate data into a table through select statement over another table provided another table has a set of fields, which are required to populate first table. Here is the syntax:

```
INSERT INTO first_table_name [(column1, column2, ... columnN)]
   SELECT column1, column2, ...columnN
   FROM second_table_name
   [WHERE condition];
```

For now, you can skip above statement, first let's learn SELECT and WHERE clauses, which will be covered in subsequent chapters.

CHAPTER

12

SQLite Select Query

SQLite **SELECT** statement is used to fetch the data from a SQLite database table which returns data in the form of result table. These result tables are also called result-sets.

Syntax:

The basic syntax of SQLite SELECT statement is as follows:

```
SELECT column1, column2, columnN FROM table_name;
```

Here, column1, column2...are the fields of a table, whose values you want to fetch. If you want to fetch all the fields available in the field, then you can use the following syntax:

```
SELECT * FROM table_name;
```

Example:

Consider COMPANY table is having following records:

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

Following is an example to fetch and display all these records using SELECT statement. Here, first three commands have been used to set properly formatted output.

```
sqlite>.header on
sqlite>.mode column
sqlite> SELECT * FROM COMPANY;
```

Finally, you will get the following result:

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
```

```
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

If you want to fetch only selected fields of COMPANY table, then use the following query:

```
sqlite> SELECT ID, NAME, SALARY FROM COMPANY;
```

Above query will produce the following result:

```
ID          NAME        SALARY
----------  ----------  ----------
1           Paul        20000.0
2           Allen       15000.0
3           Teddy       20000.0
4           Mark        65000.0
5           David       85000.0
6           Kim         45000.0
7           James       10000.0
```

Setting output column width:

Sometimes, you will face a problem related to truncated output in case of **.mode column** which happens because of default width of the column to be displayed. What you can do is that you can set column displayable column width using **.width num, num....** command as follows:

```
sqlite>.width 10, 20, 10
sqlite>SELECT * FROM COMPANY;
```

Above **.width** command sets first column width to 10, second column width to 20 and third column width to 10. So finally above SELECT statement will give the following result:

```
ID          NAME                  AGE         ADDRESS     SALARY
----------  --------------------  ----------  ----------  ----------
1           Paul                  32          California  20000.0
2           Allen                 25          Texas       15000.0
3           Teddy                 23          Norway      20000.0
4           Mark                  25          Rich-Mond   65000.0
5           David                 27          Texas       85000.0
6           Kim                   22          South-Hall  45000.0
7           James                 24          Houston     10000.0
```

Schema Information:

Because all the **dot commads** are available at SQLite prompt only, so while doing your programming with SQLite, you will use the following statement to list down all the tables created in your database using the following SELECT statement with **sqlite_master** table:

```
sqlite> SELECT tbl_name FROM sqlite_master WHERE type = 'table';
```

Assuming you have only COMPANY table in your testDB.db, this will produce the following result:

```
tbl_name
----------
```

```
COMPANY
```

You can list down complete information about COMPANY table as follows:

```
sqlite> SELECT sql FROM sqlite_master WHERE type = 'table' AND tbl_name =
'COMPANY';
```

Assuming you have only COMPANY table in your testDB.db, this will produce the following result:

```
CREATE TABLE COMPANY(
   ID INT PRIMARY KEY     NOT NULL,
   NAME           TEXT    NOT NULL,
   AGE            INT     NOT NULL,
   ADDRESS        CHAR(50),
   SALARY         REAL
)
```

CHAPTER

13

SQLite Operators

What is an operator in SQLite?

An operator is a reserved word or a character used primarily in an SQLite statement's WHERE clause to perform operation(s), such as comparisons and arithmetic operations.

Operators are used to specify conditions in an SQLite statement and to serve as conjunctions for multiple conditions in a statement.

- Arithmetic operators
- Comparison operators
- Logical operators
- Bitwise operators

SQLite Arithmetic Operators:

Assume variable a holds 10 and variable b holds 20, then:

Operator	Description	Example
+	Addition - Adds values on either side of the operator	a + b will give 30
-	Subtraction - Subtracts right hand operand from left hand operand	a - b will give -10
*	Multiplication - Multiplies values on either side of the operator	a * b will give 200
/	Division - Divides left hand operand by right hand operand	b / a will give 2
%	Modulus - Divides left hand operand by right hand operand and returns remainder	b % a will give 0

Example

Here are simple examples showing usage of SQLite Arithmetic Operators:

```
sqlite> .mode line
sqlite> select 10 + 20;
10 + 20 = 30


sqlite> select 10 - 20;
10 - 20 = -10


sqlite> select 10 * 20;
10 * 20 = 200


sqlite> select 10 / 5;
10 / 5 = 2


sqlite> select 12 %  5;
12 %  5 = 2
```

SQLite Comparison Operators:

Assume variable a holds 10 and variable b holds 20, then:

Operator	Description	Example
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(a == b) is not true.
=	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(a = b) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(a != b) is true.
<>	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(a <> b) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(a > b) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(a < b) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(a >= b) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(a <= b) is true.
!<	Checks if the value of left operand is not less than the value of right operand, if yes then condition becomes true.	(a !< b) is false.
!>	Checks if the value of left operand is not greater than the value of right operand, if yes	(a !> b)

	then condition becomes true.	is true.

Example

Consider COMPANY table has the following records:

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

Below example will show the usage of various SQLite Comparison Operators.

Here, we have used **WHERE** clause, which will be explained in a separate chapter but for now you can understand that WHERE clause is used to put a conditional statement along with SELECT statement.

Following SELECT statement lists down all the records having SALARY greater than 50,000.00:

```
sqlite> SELECT * FROM COMPANY WHERE SALARY > 50000;
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

Following SELECT statement lists down all the records having SALARY equal to 20,000.00:

```
sqlite>  SELECT * FROM COMPANY WHERE SALARY = 20000;
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
3           Teddy       23          Norway      20000.0
```

Following SELECT statement lists down all the records having SALARY not equal to 20,000.00:

```
sqlite>  SELECT * FROM COMPANY WHERE SALARY != 20000;
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
2           Allen       25          Texas       15000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

Following SELECT statement lists down all the records having SALARY not equal to 20,000.00:

```
sqlite> SELECT * FROM COMPANY WHERE SALARY <> 20000;
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
2           Allen       25          Texas       15000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

```
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

Following SELECT statement lists down all the records having SALARY greater than or equal to 65,000.00:

```
sqlite> SELECT * FROM COMPANY WHERE SALARY >= 65000;
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

SQLite Logical Operators:

Here is a list of all the logical operators available in SQLite.

Operator	Description
AND	The AND operator allows the existence of multiple conditions in an SQL statement's WHERE clause.
BETWEEN	The BETWEEN operator is used to search for values that are within a set of values, given the minimum value and the maximum value.
EXISTS	The EXISTS operator is used to search for the presence of a row in a specified table that meets certain criteria.
IN	The IN operator is used to compare a value to a list of literal values that have been specified.
NOT IN	The negation of IN operator which is used to compare a value to a list of literal values that have been specified.
LIKE	The LIKE operator is used to compare a value to similar values using wildcard operators.
GLOB	The GLOB operator is used to compare a value to similar values using wildcard operators. Also, GLOB is case sensitive, unlike LIKE.
NOT	The NOT operator reverses the meaning of the logical operator with which it is used. Eg. NOT EXISTS, NOT BETWEEN, NOT IN, etc. **This is negate operator.**
OR	The OR operator is used to combine multiple conditions in an SQL statement's WHERE clause.
IS NULL	The NULL operator is used to compare a value with a NULL value.
IS	The IS operator work like =
IS NOT	The IS operator work like !=
||	Adds two different strings and make new one.
UNIQUE	The UNIQUE operator searches every row of a specified table for uniqueness (no duplicates).

Example

Consider COMPANY table has the following records:

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
```

```
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

Here are simple examples showing usage of SQLite Logical Operators. Following SELECT statement lists down all the records where AGE is greater than or equal to 25 and salary is greater than or equal to 65000.00:

```
sqlite> SELECT * FROM COMPANY WHERE AGE >= 25 AND SALARY >= 65000;
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

Following SELECT statement lists down all the records where AGE is greater than or equal to 25 **OR** salary is greater than or equal to 65000.00:

```
sqlite> SELECT * FROM COMPANY WHERE AGE >= 25 OR SALARY >= 65000;
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

Following SELECT statement lists down all the records where AGE is not NULL which means all the records because none of the record is having AGE equal to NULL:

```
sqlite>  SELECT * FROM COMPANY WHERE AGE IS NOT NULL;
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

Following SELECT statement lists down all the records where NAME starts with 'Ki', does not matter what comes after 'Ki'.

```
sqlite> SELECT * FROM COMPANY WHERE NAME LIKE 'Ki%';
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
6           Kim         22          South-Hall  45000.0
```

Following SELECT statement lists down all the records where NAME starts with 'Ki', does not matter what comes after 'Ki':

```
sqlite> SELECT * FROM COMPANY WHERE NAME GLOB 'Ki*';
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
6           Kim         22          South-Hall  45000.0
```

Following SELECT statement lists down all the records where AGE value is either 25 or 27:

```
sqlite> SELECT * FROM COMPANY WHERE AGE IN ( 25, 27 );
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
```

```
2           Allen       25          Texas       15000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

Following SELECT statement lists down all the records where AGE value is neither 25 nor 27:

```
sqlite> SELECT * FROM COMPANY WHERE AGE NOT IN ( 25, 27 );
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
3           Teddy       23          Norway      20000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

Following SELECT statement lists down all the records where AGE value is in BETWEEN 25 AND 27:

```
sqlite> SELECT * FROM COMPANY WHERE AGE BETWEEN 25 AND 27;
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
2           Allen       25          Texas       15000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

Following SELECT statement makes use of SQL sub-query where sub-query finds all the records with AGE field having SALARY > 65000 and later WHERE clause is being used along with EXISTS operator to list down all the records where AGE from the outside query exists in the result returned by sub-query:

```
sqlite> SELECT AGE FROM COMPANY
        WHERE EXISTS (SELECT AGE FROM COMPANY WHERE SALARY > 65000);
AGE
----------
32
25
23
25
27
22
24
```

Following SELECT statement makes use of SQL sub-query where subquery finds all the records with AGE field having SALARY > 65000 and later WHERE clause is being used along with > operator to list down all the records where AGE from outside query is greater than the age in the result returned by sub-query:

```
sqlite> SELECT * FROM COMPANY
        WHERE AGE > (SELECT AGE FROM COMPANY WHERE SALARY > 65000);
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
```

SQLite Bitwise Operators:

Bitwise operator works on bits and perform bit-by-bit operation. The truth table for & and | is as follows:

P	**Q**	**p & q**	**p | q**
0	0	0	0
0	1	0	1

1	1	1	1
1	0	0	1

Assume if A = 60; and B = 13; now in binary format, they will be as follows:

A = 0011 1100

B = 0000 1101

A&B = 0000 1100

A|B = 0011 1101

~A = 1100 0011

The Bitwise operators supported by SQLite language are listed in the following table. Assume variable A holds 60 and variable B holds 13, then:

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) will give 61 which is 0011 1101
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -60 which is 1100 0011
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 0000 1111

Example

Here are simple examples showing usage of SQLite Bitwise Operators:

```
sqlite> .mode line
sqlite> select 60 | 13;
60 | 13 = 61

sqlite> select 60 & 13;
60 & 13 = 12

sqlite> select  60 ^ 13;
10 * 20 = 200


sqlite>  select  (~60);
(~60) = -61

sqlite>  select  (60 << 2);
```

```
(60 << 2) = 240

sqlite>  select  (60 >> 2);
(60 >> 2) = 15
```

CHAPTER

14

SQLite Expressions

An expression is a combination of one or more values, operators and SQL functions that evaluate to a value.

SQL EXPRESSIONs are like formulas and they are written in query language. You can also use to query the database for specific set of data.

Syntax:

Consider the basic syntax of the SELECT statement as follows:

```
SELECT column1, column2, columnN
FROM table_name
WHERE [CONTION | EXPRESSION];
```

There are different types of SQLite expressions, which are mentioned below:

SQLite - Boolean Expressions:

SQLite Boolean Expressions fetch the data on the basis of matching single value. Following is the syntax:

```
SELECT column1, column2, columnN
FROM table_name
WHERE SINGLE VALUE MATCHTING EXPRESSION;
```

Consider COMPANY table has the following records:

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

Here is simple examples showing usage of SQLite Boolean Expressions:

```
sqlite> SELECT * FROM COMPANY WHERE SALARY = 10000;
```

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
4           James        24          Houston   10000.0
```

SQLite - Numeric Expression:

These expressions are used to perform any mathematical operation in any query. Following is the syntax:

```
SELECT numerical_expression as  OPERATION_NAME
[FROM table_name WHERE CONDITION] ;
```

Here, numerical_expression is used for mathematical expression or any formula. Following is a simple example showing usage of SQLite Numeric Expressions:

```
sqlite> SELECT (15 + 6) AS ADDITION
ADDITION = 21
```

There are several built-in functions like avg(), sum(), count(), etc., to perform what is known as aggregate data calculations against a table or a specific table column.

```
sqlite> SELECT COUNT(*) AS "RECORDS" FROM COMPANY;
RECORDS = 7
```

SQLite - Date Expressions:

Date Expressions return current system date and time values and these expressions will be used in various data manipulations.

```
sqlite>  SELECT CURRENT_TIMESTAMP;
CURRENT_TIMESTAMP = 2013-03-17 10:43:35
```

CHAPTER

15

SQLite Where Clause

The SQLite **WHERE** clause is used to specify a condition while fetching the data from one table or multiple tables.

If the given condition is satisfied, means true, then it returns specific value from the table. You would use WHERE clause to filter the records and fetching only necessary records.

The WHERE clause not only used in SELECT statement, but it is also used in UPDATE, DELETE statement, etc., which we would study in subsequent chapters.

Syntax:

The basic syntax of SQLite SELECT statement with WHERE clause is as follows:

```
SELECT column1, column2, columnN
FROM table_name
WHERE [condition]
```

Example:

You can specify a condition using Comparision or Logical Operators like >, <, =, LIKE, NOT, etc. Consider COMPANY table has the following records:

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

Here are simple examples showing usage of SQLite Logical Operators. Following SELECT statement lists down all the records where AGE is greater than or equal to 25 **AND** salary is greater than or equal to 65000.00:

```
sqlite> SELECT * FROM COMPANY WHERE AGE >= 25 AND SALARY >= 65000;
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
4           Mark        25          Rich-Mond   65000.0
```

```
5           David       27          Texas       85000.0
```

Following SELECT statement lists down all the records where AGE is greater than or equal to 25 **OR** salary is greater than or equal to 65000.00:

```
sqlite> SELECT * FROM COMPANY WHERE AGE >= 25 OR SALARY >= 65000;
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

Following SELECT statement lists down all the records where AGE is not NULL which means all the records because none of the record is having AGE equal to NULL:

```
sqlite>  SELECT * FROM COMPANY WHERE AGE IS NOT NULL;
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

Following SELECT statement lists down all the records where NAME starts with 'Ki', does not matter what comes after 'Ki'.

```
sqlite> SELECT * FROM COMPANY WHERE NAME LIKE 'Ki%';
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
6           Kim         22          South-Hall  45000.0
```

Following SELECT statement lists down all the records where NAME starts with 'Ki', does not matter what comes after 'Ki':

```
sqlite> SELECT * FROM COMPANY WHERE NAME GLOB 'Ki*';
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
6           Kim         22          South-Hall  45000.0
```

Following SELECT statement lists down all the records where AGE value is either 25 or 27:

```
sqlite> SELECT * FROM COMPANY WHERE AGE IN ( 25, 27 );
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
2           Allen       25          Texas       15000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

Following SELECT statement lists down all the records where AGE value is neither 25 nor 27:

```
sqlite> SELECT * FROM COMPANY WHERE AGE NOT IN ( 25, 27 );
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
3           Teddy       23          Norway      20000.0
6           Kim         22          South-Hall  45000.0
```

```
7           James       24          Houston     10000.0
```

Following SELECT statement lists down all the records where AGE value is in BETWEEN 25 AND 27:

```
sqlite> SELECT * FROM COMPANY WHERE AGE BETWEEN 25 AND 27;
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
2           Allen       25          Texas       15000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

Following SELECT statement makes use of SQL sub-query where sub-query finds all the records with AGE field having SALARY > 65000 and later WHERE clause is being used along with EXISTS operator to list down all the records where AGE from the outside query exists in the result returned by sub-query:

```
sqlite> SELECT AGE FROM COMPANY
        WHERE EXISTS (SELECT AGE FROM COMPANY WHERE SALARY > 65000);
AGE
----------
32
25
23
25
27
22
24
```

Following SELECT statement makes use of SQL sub-query where sub-query finds all the records with AGE field having SALARY > 65000 and later WHERE clause is being used along with > operator to list down all the records where AGE from outside query is greater than the age in the result returned by sub-query:

```
sqlite> SELECT * FROM COMPANY
        WHERE AGE > (SELECT AGE FROM COMPANY WHERE SALARY > 65000);
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
```

CHAPTER

16

SQLite AND and OR Operator

The SQLite **AND** and **OR** operators are used to combine multiple conditions to narrow down selected data in an SQLite statement. These two operators are called conjunctive operators.

These operators provide a means to make multiple comparisons with different operators in the same SQLite statement.

The AND Operator:

The **AND** operator allows the existence of multiple conditions in an SQLite statement's WHERE clause. While using AND operator, complete condition will be assumed true when all the conditions are true. For example, [condition1] AND [condition2] will be true only when both condition1 and condition2 are true.

Syntax:

The basic syntax of AND operator with WHERE clause is as follows:

```
SELECT column1, column2, columnN
FROM table_name
WHERE [condition1] AND [condition2]...AND [conditionN];
```

You can combine N number of conditions using AND operator. For an action to be taken by the SQLite statement, whether it be a transaction or query, all conditions separated by the AND must be TRUE.

Example:

Consider COMPANY table is having the following records:

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

Following SELECT statement lists down all the records where AGE is greater than or equal to 25 **AND**salary is greater than or equal to 65000.00:

```
sqlite> SELECT * FROM COMPANY WHERE AGE >= 25 AND SALARY >= 65000;
```

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

The OR Operator:

The OR operator is also used to combine multiple conditions in an SQLite statement's WHERE clause. While using OR operator, complete condition will be assumed true when at least any of the conditions is true. For example, [condition1] OR [condition2] will be true if either condition1 or condition2 is true.

Syntax:

The basic syntax of OR operator with WHERE clause is as follows:

```
SELECT column1, column2, columnN
FROM table_name
WHERE [condition1] OR [condition2]...OR [conditionN]
```

You can combine N number of conditions using OR operator. For an action to be taken by the SQLite statement, whether it be a transaction or query, only any ONE of the conditions separated by the OR must be TRUE.

Example:

Consider COMPANY table is having the following records:

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

Following SELECT statement lists down all the records where AGE is greater than or equal to 25 **OR**salary is greater than or equal to 65000.00:

```
sqlite> SELECT * FROM COMPANY WHERE AGE >= 25 OR SALARY >= 65000;
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

CHAPTER

17

SQLite Update Query

The SQLite **UPDATE** Query is used to modify the existing records in a table. You can use WHERE clause with UPDATE query to update selected rows, otherwise all the rows would be updated.

Syntax:

The basic syntax of UPDATE query with WHERE clause is as follows:

```
UPDATE table_name
SET column1 = value1, column2 = value2...., columnN = valueN
WHERE [condition];
```

You can combine N number of conditions using AND or OR operators.

Example:

Consider COMPANY table is having the following records:

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

Following is an example, which would update ADDRESS for a customer whose ID is 6:

```
sqlite> UPDATE COMPANY SET ADDRESS = 'Texas' WHERE ID = 6;
```

Now, COMPANY table would have the following records:

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

```
6           Kim         22          Texas       45000.0
7           James       24          Houston     10000.0
```

If you want to modify all ADDRESS and SALARY column values in COMPANY table, you do not need to use WHERE clause and UPDATE query would be as follows:

```
sqlite> UPDATE COMPANY SET ADDRESS = 'Texas', SALARY = 20000.00;
```

Now, COMPANY table will have the following records:

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          Texas       20000.0
2           Allen       25          Texas       20000.0
3           Teddy       23          Texas       20000.0
4           Mark        25          Texas       20000.0
5           David       27          Texas       20000.0
6           Kim         22          Texas       20000.0
7           James       24          Texas       20000.0
```

CHAPTER

18

SQLite Delete Query

The SQLite **DELETE** Query is used to delete the existing records from a table. You can use WHERE clause with DELETE query to delete selected rows, otherwise all the records would be deleted.

Syntax:

The basic syntax of DELETE query with WHERE clause is as follows:

```
DELETE FROM table_name
WHERE [condition];
```

You can combine N number of conditions using AND or OR operators.

Example:

Consider COMPANY table is having the following records:

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

Following is an example, which would DELETE a customer whose ID is 7:

```
sqlite> DELETE FROM COMPANY WHERE ID = 7;
```

Now, COMPANY table will have the following records:

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
```

```
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
```

If you want to DELETE all the records from COMPANY table, you do not need to use WHERE clause with DELETE query, which would be as follows:

```
sqlite> DELETE FROM COMPANY;
```

Now, COMPANY table does not have any record because all the records have been deleted by DELETE statement.

CHAPTER

19

SQLite Like Clause

The SQLite **LIKE** operator is used to match text values against a pattern using wildcards. If the search expression can be matched to the pattern expression, the LIKE operator will return true, which is 1. There are two wildcards used in conjunction with the LIKE operator:

- The percent sign (%)
- The underscore (_)

The percent sign represents zero, one, or multiple numbers or characters. The underscore represents a single number or character. These symbols can be used in combinations.

Syntax:

The basic syntax of % and _ is as follows:

```
SELECT FROM table_name
WHERE column LIKE 'XXXX%'

or

SELECT FROM table_name
WHERE column LIKE '%XXXX%'

or

SELECT FROM table_name
WHERE column LIKE 'XXXX_'

or

SELECT FROM table_name
WHERE column LIKE '_XXXX'

or

SELECT FROM table_name
WHERE column LIKE '_XXXX_'
```

You can combine N number of conditions using AND or OR operators. Here XXXX could be any numeric or string value.

Example:

Here are number of examples showing WHERE part having different LIKE clause with '%' and '_' operators:

Statement	Description
WHERE SALARY LIKE '200%'	Finds any values that start with 200
WHERE SALARY LIKE '%200%'	Finds any values that have 200 in any position
WHERE SALARY LIKE '_00%'	Finds any values that have 00 in the second and third positions
WHERE SALARY LIKE '2_%_%'	Finds any values that start with 2 and are at least 3 characters in length
WHERE SALARY LIKE '%2'	Finds any values that end with 2
WHERE SALARY LIKE '_2%3'	Finds any values that have a 2 in the second position and end with a 3
WHERE SALARY LIKE '2___3'	Finds any values in a five-digit number that start with 2 and end with 3

Let us take a real example, consider COMPANY table is having the following records:

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

Following is an example, which would display all the records from COMPANY table where AGE starts with 2:

```
sqlite> SELECT * FROM COMPANY WHERE AGE  LIKE '2%';
```

This would produce the following result:

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

Following is an example, which would display all the records from COMPANY table where ADDRESS will have a hyphen (-) inside the text:

```
sqlite> SELECT * FROM COMPANY WHERE ADDRESS  LIKE '%-%';
```

This would produce the following result:

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
4           Mark        25          Rich-Mond   65000.0
6           Kim         22          South-Hall  45000.0
```

CHAPTER

20

SQLite Glob Clause

The SQLite **GLOB** operator is used to match only text values against a pattern using wildcards. If the search expression can be matched to the pattern expression, the GLOB operator will return true, which is 1. Unlike LIKE operator, GLOB is case sensitive and it follows syntax of UNIX for specifying the following wildcards.

- The asterisk sign (*)
- The question mark (?)

The asterisk sign represents zero or multiple numbers or characters. The ? represents a single number or character.

Syntax:

The basic syntax of * and ? is as follows:

```
SELECT FROM table_name
WHERE column GLOB 'XXXX*'

or

SELECT FROM table_name
WHERE column GLOB '*XXXX*'

or

SELECT FROM table_name
WHERE column GLOB 'XXXX?'

or

SELECT FROM table_name
WHERE column GLOB '?XXXX'

or

SELECT FROM table_name
WHERE column GLOB '?XXXX?'

or
```

```
SELECT FROM table_name
WHERE column GLOB '????'
```

You can combine N number of conditions using AND or OR operators. Here XXXX could be any numberic or string value.

Example:

Here are number of examples showing WHERE part having different LIKE clause with '*' and '?' operators:

Statement	Description
WHERE SALARY GLOB '200*'	Finds any values that start with 200
WHERE SALARY GLOB '*200*'	Finds any values that have 200 in any position
WHERE SALARY GLOB '?00*'	Finds any values that have 00 in the second and third positions
WHERE SALARY GLOB '2??'	Finds any values that start with 2 and are at least 3 characters in length
WHERE SALARY GLOB '*2'	Finds any values that end with 2
WHERE SALARY GLOB '?2*3'	Finds any values that have a 2 in the second position and end with a 3
WHERE SALARY GLOB '2???3'	Finds any values in a five-digit number that start with 2 and end with 3

Let us take a real example, consider COMPANY table is having the following records:

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

Following is an example, which would display all the records from COMPANY table where AGE starts with 2:

```
sqlite> SELECT * FROM COMPANY WHERE AGE  GLOB '2*';
```

This would produce following result:

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

Following is an example, which would display all the records from COMPANY table where ADDRESS will have a hyphen (-) inside the text:

```
sqlite> SELECT * FROM COMPANY WHERE ADDRESS  GLOB '*-*';
```

This would produce the following result:

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
4           Mark        25          Rich-Mond   65000.0
6           Kim         22          South-Hall  45000.0
```

CHAPTER

21

SQLite LIMIT Clause

The SQLite **LIMIT** clause is used to limit the data amount returned by the SELECT statement.

Syntax:

The basic syntax of SELECT statement with LIMIT clause is as follows:

```
SELECT column1, column2, columnN
FROM table_name
LIMIT [no of rows]
```

Following is the syntax of LIMIT clause when it is used along with OFFSET clause:

```
SELECT column1, column2, columnN
FROM table_name
LIMIT [no of rows] OFFSET [row num]
```

SQLite engine will return rows starting from the next row to the given OFFSET as shown below in the last example.

Example:

Consider COMPANY table is having the following records:

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

Following is an example, which limits the row in the table according to the number of rows you want to fetch from table:

```
sqlite> SELECT * FROM COMPANY LIMIT 6;
```

This would produce the following result:

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
```

But in certain situations, you may need to pick up a set of records from a particular offset. Here is an example, which picks up 3 records starting from 3rd position:

```
sqlite> SELECT * FROM COMPANY LIMIT 3 OFFSET 2;
```

This would produce the following result:

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

CHAPTER

22

SQLite Order By Clause

The SQLite **ORDER BY** clause is used to sort the data in ascending or descending order, based on one or more columns.

Syntax:

The basic syntax of ORDER BY clause is as follows:

```
SELECT column-list
FROM table_name
[WHERE condition]
[ORDER BY column1, column2, .. columnN] [ASC | DESC];
```

You can use more than one column in the ORDER BY clause. Make sure whatever column you are using to sort, that column should be available in column-list.

Example:

Consider COMPANY table is having the following records:

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

Following is an example, which would sort the result in descending order by SALARY:

```
sqlite> SELECT * FROM COMPANY ORDER BY SALARY ASC;
```

This would produce the following result:

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
7           James       24          Houston     10000.0
2           Allen       25          Texas       15000.0
1           Paul        32          California  20000.0
```

```
3           Teddy       23          Norway      20000.0
6           Kim         22          South-Hall  45000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

Following is an example, which would sort the result in descending order by NAME and SALARY:

```
sqlite> SELECT * FROM COMPANY ORDER BY NAME, SALARY ASC;
```

This would produce following result:

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
2           Allen       25          Texas       15000.0
5           David       27          Texas       85000.0
7           James       24          Houston     10000.0
6           Kim         22          South-Hall  45000.0
4           Mark        25          Rich-Mond   65000.0
1           Paul        32          California  20000.0
3           Teddy       23          Norway      20000.0
```

Following is an example, which would sort the result in descending order by NAME:

```
sqlite> SELECT * FROM COMPANY ORDER BY NAME DESC;
```

This would produce the following result:

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
3           Teddy       23          Norway      20000.0
1           Paul        32          California  20000.0
4           Mark        25          Rich-Mond   65000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
5           David       27          Texas       85000.0
2           Allen       25          Texas       15000.0
```

CHAPTER 23

SQLite Group By Clause

The SQLite **GROUP BY** clause is used in collaboration with the SELECT statement to arrange identical data into groups.

The GROUP BY clause follows the WHERE clause in a SELECT statement and precedes the ORDER BY clause.

Syntax:

The basic syntax of GROUP BY clause is given below. The GROUP BY clause must follow the conditions in the WHERE clause and must precede the ORDER BY clause if one is used.

```
SELECT column-list
FROM table_name
WHERE [ conditions ]
GROUP BY column1, column2....columnN
ORDER BY column1, column2....columnN
```

You can use more than one column in the GROUP BY clause. Make sure whatever column you are using to group, that column should be available in column-list.

Example:

Consider COMPANY table is having the following records:

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

If you want to know the total amount of salary on each customer, then GROUP BY query would be as follows:

```
sqlite> SELECT NAME, SUM(SALARY) FROM COMPANY GROUP BY NAME;
```

This would produce following result:

```
NAME        SUM(SALARY)
----------  -----------
Allen       15000.0
David       85000.0
James       10000.0
Kim         45000.0
Mark        65000.0
Paul        20000.0
Teddy       20000.0
```

Now, let us create three more records in COMPANY table using the following INSERT statements:

```
INSERT INTO COMPANY VALUES (8, 'Paul', 24, 'Houston', 20000.00 );
INSERT INTO COMPANY VALUES (9, 'James', 44, 'Norway', 5000.00 );
INSERT INTO COMPANY VALUES (10, 'James', 45, 'Texas', 5000.00 );
```

Now, our table has the following records with duplicate names:

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
8           Paul        24          Houston     20000.0
9           James       44          Norway      5000.0
10          James       45          Texas       5000.0
```

Again, let us use the same statement to group-by all the records using NAME column as follows:

```
sqlite> SELECT NAME, SUM(SALARY) FROM COMPANY GROUP BY NAME ORDER BY NAME;
```

This would produce the following result:

```
NAME        SUM(SALARY)
----------  -----------
Allen       15000
David       85000
James       20000
Kim         45000
Mark        65000
Paul        40000
Teddy       20000
```

Let us use ORDER BY clause along with GROUP BY clause as follows:

```
sqlite>  SELECT NAME, SUM(SALARY)
         FROM COMPANY GROUP BY NAME ORDER BY NAME DESC;
```

This would produce the following result:

```
NAME        SUM(SALARY)
----------  -----------
Teddy       20000
Paul        40000
Mark        65000
Kim         45000
```

```
James       20000
David       85000
Allen       15000
```

CHAPTER

24

SQLite Having Clause

The HAVING clause enables you to specify conditions that filter which group results appear in the final results.

The WHERE clause places conditions on the selected columns, whereas the HAVING clause places conditions on groups created by the GROUP BY clause.

Syntax:

The following is the position of the HAVING clause in a SELECT query:

```
SELECT
FROM
WHERE
GROUP BY
HAVING
ORDER BY
```

The HAVING clause must follow the GROUP BY clause in a query and must also precede the ORDER BY clause if used. The following is the syntax of the SELECT statement, including the HAVING clause:

```
SELECT column1, column2
FROM table1, table2
WHERE [ conditions ]
GROUP BY column1, column2
HAVING [ conditions ]
ORDER BY column1, column2
```

Example:

Consider COMPANY table is having the following records:

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
```

```
7           James       24          Houston     10000.0
8           Paul        24          Houston     20000.0
9           James       44          Norway      5000.0
10          James       45          Texas       5000.0
```

Following is the example, which would display record for which name count is less than 2:

```
sqlite > SELECT * FROM COMPANY GROUP BY name HAVING count(name) < 2;
```

This would produce the following result:

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
2           Allen       25          Texas       15000
5           David       27          Texas       85000
6           Kim         22          South-Hall  45000
4           Mark        25          Rich-Mond   65000
3           Teddy       23          Norway      20000
```

Following is the example, which would display record for which name count is greater than 2:

```
sqlite > SELECT * FROM COMPANY GROUP BY name HAVING count(name) > 2;
```

This would produce the following result:

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
10          James       45          Texas       5000
```

CHAPTER

25

SQLite Distinct Keyword

The SQLite **DISTINCT** keyword is used in conjunction with SELECT statement to eliminate all the duplicate records and fetching only unique records.

There may be a situation when you have multiple duplicate records in a table. While fetching such records, it makes more sense to fetch only unique records instead of fetching duplicate records.

Syntax:

The basic syntax of DISTINCT keyword to eliminate duplicate records is as follows:

```
SELECT DISTINCT column1, column2,.....columnN
FROM table_name
WHERE [condition]
```

Example:

Consider COMPANY table is having the following records:

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
8           Paul        24          Houston     20000.0
9           James       44          Norway      5000.0
10          James       45          Texas       5000.0
```

First, let us see how the following SELECT query returns duplicate salary records:

```
sqlite> SELECT name FROM COMPANY;
```

This would produce the following result:

```
NAME
----------
```

```
Paul
Allen
Teddy
Mark
David
Kim
James
Paul
James
James
```

Now, let us use **DISTINCT** keyword with the above SELECT query and see the result:

```
sqlite> SELECT DISTINCT name FROM COMPANY;
```

This would produce the following result, where we do not have any duplicate entry:

```
NAME
----------
Paul
Allen
Teddy
Mark
David
Kim
James
```

CHAPTER

26

SQLite PRAGMA

The SQLite **PRAGMA** command is a special command to be used to control various environmental variables and state flags within the SQLite environment. A PRAGMA value can be read and it can also be set based on requirements.

Syntax:

To query the current PRAGMA value, just provide the name of the pragma:

```
PRAGMA pragma_name;
```

To set a new value for PRAGMA, you will use the following syntax:

```
PRAGMA pragma_name = value;
```

The set mode can be either the name or the integer equivalent but the returned value will always be an integer.

auto_vacuum Pragma

The **auto_vacuum** pragma gets or sets the auto-vacuum mode. Following is the simple syntax:

```
PRAGMA [database.]auto_vacuum;
PRAGMA [database.]auto_vacuum = mode;
```

Where **mode** can be any of the following:

Pragma Value	Description
0 or NONE	Auto-vacuum is disabled. This is default mode which means that a database file will never shrink in size unless it is manually vacuumed using the VACUUM command.
1 or FULL	Auto-vacuum is enabled and fully automatic which allows a database file to shrink as data is removed from the database.
2 or INCREMENTAL	Auto-vacuum is enabled but must be manually activated. In this mode the reference data is maintained, but free pages are simply put on the free list. These pages can be recovered using the **incremental_vacuum pragma**any time.

cache_size Pragma

The **cache_size** pragma can get or temporarily set the maximum size of the in-memory page cache. Following is the simple syntax:

```
PRAGMA [database.]cache_size;
PRAGMA [database.]cache_size = pages;
```

The **pages** value represents the number of pages in the cache. The built-in page cache has a default size of 2,000 pages and a minimum size of 10 pages.

case_sensitive_like Pragma

The **case_sensitive_like** pragma controls the case-sensitivity of the built-in LIKE expression. By default, this pragma is false which means that the built-in LIKE operator ignores letter case. Following is the simple syntax:

```
PRAGMA case_sensitive_like = [true|false];
```

There is no way to query for the current state of this pragma.

count_changes Pragma

The **count_changes** pragma gets or sets the return value of data manipulation statements such as INSERT, UPDATE and DELETE. Following is the simple syntax:

```
PRAGMA count_changes;
PRAGMA count_changes = [true|false];
```

By default, this pragma is false and these statements do not return anything. If set to true, each of the mentioned statement will return an one-column, one-row table consisting of a single integer value indicating impacted rows by the operation.

database_list Pragma

The **database_list** pragma will be used to list down all the databases attached. Following is the simple syntax:

```
PRAGMA database_list;
```

This pragma will return a three-column table with one row per open or attached database giving database sequence number, its name and file associated.

encoding Pragma

The **encoding** pragma controls how strings are encoded and stored in a database file. Following is the simple syntax:

```
PRAGMA encoding;
PRAGMA encoding = format;
```

The format value can be one of UTF-8, UTF-16le, or UTF-16be.

freelist_count Pragma

The **freelist_count** pragma returns a single integer indicating how many database pages are currently marked as free and available. Following is the simple syntax:

```
PRAGMA [database.]freelist_count;
```

The format value can be one of UTF-8, UTF-16le, or UTF-16be.

index_info Pragma

The **index_info** pragma returns information about a database index. Following is the simple syntax:

```
PRAGMA [database.]index_info( index_name );
```

The result set will contain one row for each column contained in the index giving column sequence, column index within table and column name.

index_list Pragma

The **index_list** pragma lists all of the indexes associated with a table. Following is the simple syntax:

```
PRAGMA [database.]index_list( table_name );
```

The result set will contain one row for each index giving index sequence, index name and flag indicating whether index is unique or not.

journal_mode Pragma

The **journal_mode** pragma gets or sets the journal mode which controls how the journal file is stored and processed. Following is the simple syntax:

```
PRAGMA journal_mode;
PRAGMA journal_mode = mode;
PRAGMA database.journal_mode;
PRAGMA database.journal_mode = mode;
```

There are five supported journal modes:

Pragma Value	Description
DELETE	This is default mode. Here at the conclusion of a transaction, the journal file is deleted.
TRUNCATE	The journal file is truncated to a length of zero bytes.
PERSIST	The journal file is left in place, but the header is overwritten to indicate the journal is no longer valid.
MEMORY	The journal record is held in memory, rather than on disk.
OFF	No journal record is kept.

max_page_count Pragma

The **max_page_count** pragma gets or sets the maximum allowed page count for a database. Following is the simple syntax:

```
PRAGMA [database.]max_page_count;
PRAGMA [database.]max_page_count = max_page;
```

The default value is 1,073,741,823 which is one giga-page which means if the default 1 KB page size, this allows databases to grow up to one terabyte.

page_count Pragma

The **page_count** pragma returns the current number of pages in database. Following is the simple syntax:

```
PRAGMA [database.]page_count;
```

The size of the database file should be page_count * page_size.

page_size Pragma

The **page_size** pragma gets or sets the size of the database pages. Following is the simple syntax:

```
PRAGMA [database.]page_size;
PRAGMA [database.]page_size = bytes;
```

By default, the allowed sizes are 512, 1024, 2048, 4096, 8192, 16384, and 32768 bytes. The only way to alter the page size on an existing database is to set the page size and then immediately VACUUM the database.

parser_trace Pragma

The **parser_trace** pragma controls printing the debugging state as it parses SQL commands. Following is the simple syntax:

```
PRAGMA parser_trace = [true|false];
```

By default, it is set to false but when enabled by setting it to true, the SQL parser will print its state as it parses SQL commands.

recursive_triggers Pragma

The **recursive_triggers** pragma gets or sets the recursive trigger functionality. If recursive triggers are not enabled, a trigger action will not fire another trigger. Following is the simple syntax:

```
PRAGMA recursive_triggers;
PRAGMA recursive_triggers = [true|false];
```

schema_version Pragma

The **schema_version** pragma gets or sets the schema version value that is stored in the database header. Following is the simple syntax:

```
PRAGMA [database.]schema_version;
PRAGMA [database.]schema_version = number;
```

This is a 32-bit signed integer value that keeps track of schema changes. Whenever a schema-altering command is executed (like, CREATE... or DROP...), this value is incremented.

secure_delete Pragma

The **secure_delete** pragma is used to control how content is deleted from the database. Following is the simple syntax:

```
PRAGMA secure_delete;
PRAGMA secure_delete = [true|false];
PRAGMA database.secure_delete;
PRAGMA database.secure_delete = [true|false];
```

The default value for the secure delete flag is normally off, but this can be changed with the SQLITE_SECURE_DELETE build option.

sql_trace Pragma

The **sql_trace** pragma is used to dump SQL trace results to the screen. Following is the simple syntax:

```
PRAGMA sql_trace;
PRAGMA sql_trace = [true|false];
```

SQLite must be compiled with the SQLITE_DEBUG directive for this pragma to be included.

synchronous Pragma

The **synchronous** pragma gets or sets the current disk synchronization mode which controls how aggressively SQLite will write data all the way out to physical storage. Following is the simple syntax:

```
PRAGMA [database.]synchronous;
PRAGMA [database.]synchronous = mode;
```

SQLite supports the following synchronisation modes:

Pragma Value	Description
0 or OFF	No syncs at all
1 or NORMAL	Sync after each sequence of critical disk operations
2 or FULL	Sync after each critical disk operation

temp_store Pragma

The **temp_store** pragma gets or sets the storage mode used by temporary database files. Following is the simple syntax:

```
PRAGMA temp_store;
PRAGMA temp_store = mode;
```

SQLite supports the following storage modes:

Pragma Value	Description
0 or DEFAULT	Use compile-time default. Normally FILE.
1 or FILE	Use file-based storage.
2 or MEMORY	Use memory-based storage.

temp_store_directory Pragma

The **temp_store_directory** pragma gets or sets the location used for temporary database files. Following is the simple syntax:

```
PRAGMA temp_store_directory;
PRAGMA temp_store_directory = 'directory_path';
```

user_version Pragma

The **user_version** pragma gets or sets the user-defined version value that is stored in the database header. Following is simple syntax:

```
PRAGMA [database.]user_version;
PRAGMA [database.]user_version = number;
```

This is a 32-bit signed integer value, which can be set by the developer for version tracking purpose.

writable_schema Pragma

The **writable_schema** pragma gets or sets the ability to modify system tables. Following is the simple syntax:

```
PRAGMA writable_schema;
PRAGMA writable_schema = [true|false];
```

If this pragma is set, tables that start with sqlite_ can be created and modified, including the sqlite_master table. Be careful while using pragma because it can lead to complete database corruption.

CHAPTER

27

SQLite Constraints

Constraints are the rules enforced on data columns on table. These are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the database.

Constraints could be column level or table level. Column level constraints are applied only to one column where as table level constraints are applied to the whole table.

Following are commonly used constraints available in SQLite.

- **NOT NULL Constraint**: Ensures that a column cannot have NULL value.
- **DEFAULT Constraint** : Provides a default value for a column when none is specified.
- **UNIQUE Constraint**: Ensures that all values in a column are different.
- **PRIMARY Key**: Uniquely identified each rows/records in a database table.
- **CHECK Constraint**: The CHECK constraint ensures that all values in a column satisfy certain conditions.

NOT NULL Constraint

By default, a column can hold NULL values. If you do not want a column to have a NULL value, then you need to define such constraint on this column specifying that NULL is now not allowed for that column.

A NULL is not the same as no data, rather, it represents unknown data.

EXAMPLE:

For example, the following SQLite statement creates a new table called COMPANY and adds five columns, three of which, ID and NAME and AGE, specify not to accept NULLs:

```
CREATE TABLE COMPANY(
   ID INT PRIMARY KEY      NOT NULL,
   NAME           TEXT     NOT NULL,
   AGE            INT      NOT NULL,
   ADDRESS        CHAR(50),
   SALARY         REAL
);
```

DEFAULT Constraint

The DEFAULT constraint provides a default value to a column when the INSERT INTO statement does not provide a specific value.

EXAMPLE:

For example, the following SQLite statement creates a new table called COMPANY and adds five columns. Here, SALARY column is set to 5000.00 by default, so in case INSERT INTO statement does not provide a value for this column, then by default, this column would be set to 5000.00.

```
CREATE TABLE COMPANY(
   ID INT PRIMARY KEY     NOT NULL,
   NAME           TEXT    NOT NULL,
   AGE            INT     NOT NULL,
   ADDRESS        CHAR(50),
   SALARY         REAL    DEFAULT 50000.00
);
```

UNIQUE Constraint

The UNIQUE Constraint prevents two records from having identical values in a particular column. In the COMPANY table, for example, you might want to prevent two or more people from having identical age.

EXAMPLE:

For example, the following SQLite statement creates a new table called COMPANY and adds five columns. Here, AGE column is set to UNIQUE, so that you can not have two records with same age:

```
CREATE TABLE COMPANY(
   ID INT PRIMARY KEY     NOT NULL,
   NAME           TEXT    NOT NULL,
   AGE            INT     NOT NULL UNIQUE,
   ADDRESS        CHAR(50),
   SALARY         REAL    DEFAULT 50000.00
);
```

PRIMARY KEY Constraint

The PRIMARY KEY constraint uniquely identifies each record in a database table. There can be more UNIQUE columns, but only one primary key in a table. Primary keys are important when designing the database tables. Primary keys are unique ids.

We use them to refer to table rows. Primary keys become foreign keys in other tables, when creating relations among tables. Due to a 'longstanding coding oversight', primary keys can be NULL in SQLite. This is not the case with other databases.

A primary key is a field in a table which uniquely identifies the each rows/records in a database table. Primary keys must contain unique values. A primary key column cannot have NULL values.

A table can have only one primary key which may consist of single or multiple fields. When multiple fields are used as a primary key, they are called a **composite key**.

If a table has a primary key defined on any field(s), then you can not have two records having the same value of that field(s).

EXAMPLE:

You already have seen various examples above where we have created COMAPNY table with ID as primary key:

```
CREATE TABLE COMPANY(
   ID INT PRIMARY KEY     NOT NULL,
```

```
   NAME           TEXT    NOT NULL,
   AGE            INT     NOT NULL,
   ADDRESS        CHAR(50),
   SALARY         REAL
);
```

CHECK Constraint

The CHECK Constraint enables a condition to check the value being entered into a record. If the condition evaluates to false, the record violates the constraint and isn't entered into the table.

EXAMPLE:

For example, the following SQLite creates a new table called COMPANY and adds five columns. Here, we add a CHECK with SALARY column, so that you can not have any SALARY Zero:

```
CREATE TABLE COMPANY3(
   ID INT PRIMARY KEY     NOT NULL,
   NAME           TEXT    NOT NULL,
   AGE            INT     NOT NULL,
   ADDRESS        CHAR(50),
   SALARY         REAL    CHECK(SALARY > 0)
);
```

Dropping Constraints:

SQLite supports a limited subset of ALTER TABLE. The ALTER TABLE command in SQLite allows the user to rename a table or to add a new column to an existing table. It is not possible to rename a column, remove a column, or add or remove constraints from a table.

CHAPTER

28

SQLite Joins

The SQLite **Joins** clause is used to combine records from two or more tables in a database. A JOIN is a means for combining fields from two tables by using values common to each.

SQL defines three major types of joins:

- The CROSS JOIN
- The INNER JOIN
- The OUTER JOIN

Before we proceed, let's consider two tables COMPANY and DEPARTMENT. We already have seen INSERT statements to populate COMPANY table. So just let's assume the list of records available in COMPANY table:

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

Another table is DEPARTMENT has the following definition:

```
CREATE TABLE DEPARTMENT(
   ID INT PRIMARY KEY      NOT NULL,
   DEPT           CHAR(50) NOT NULL,
   EMP_ID         INT      NOT NULL
);
```

Here is the list of INSERT statements to populate DEPARTMENT table:

```
INSERT INTO DEPARTMENT (ID, DEPT, EMP_ID)
VALUES (1, 'IT Billing', 1 );

INSERT INTO DEPARTMENT (ID, DEPT, EMP_ID)
VALUES (2, 'Engineering', 2 );
```

```
INSERT INTO DEPARTMENT (ID, DEPT, EMP_ID)
VALUES (3, 'Finance', 7 );
```

Finally, we have the following list of records available in DEPARTMENT table:

```
ID          DEPT        EMP_ID
----------  ----------  ----------
1           IT Billing  1
2           Engineerin  2
3           Finance     7
```

The CROSS JOIN

A CROSS JOIN matches every row of the first table with every row of the second table. If the input tables have x and y columns, respectively, the resulting table will have x+y columns. Because CROSS JOINs have the potential to generate extremely large tables, care must be taken to only use them when appropriate.

Following is the syntax of CROSS JOIN:

```
SELECT ... FROM table1 CROSS JOIN table2 ...
```

Based on the above tables, we can write a cross join as follows:

```
sqlite> SELECT EMP_ID, NAME, DEPT FROM COMPANY CROSS JOIN DEPARTMENT;
```

Above query will produce the following result:

```
EMP_ID      NAME        DEPT
----------  ----------  ----------
1           Paul        IT Billing
2           Paul        Engineerin
7           Paul        Finance
1           Allen       IT Billing
2           Allen       Engineerin
7           Allen       Finance
1           Teddy       IT Billing
2           Teddy       Engineerin
7           Teddy       Finance
1           Mark        IT Billing
2           Mark        Engineerin
7           Mark        Finance
1           David       IT Billing
2           David       Engineerin
7           David       Finance
1           Kim         IT Billing
2           Kim         Engineerin
7           Kim         Finance
1           James       IT Billing
2           James       Engineerin
7           James       Finance
```

The INNER JOIN

A INNER JOIN creates a new result table by combining column values of two tables (table1 and table2) based upon the join-predicate. The query compares each row of table1 with each row of table2 to find all pairs of rows, which satisfy the join-predicate. When the join-predicate is satisfied, column values for each matched pair of rows of A and B are combined into a result row.

An INNER JOIN is the most common type of join and is the default type of join. You can use INNER keyword optionally.

Following is the syntax of INNER JOIN:

```
SELECT ... FROM table1 [INNER] JOIN table2 ON conditional_expression ...
```

To avoid redundancy and keep the phrasing shorter, INNER JOIN conditions can be declared with a **USING** expression. This expression specifies a list of one or more columns:

```
SELECT ... FROM table1 JOIN table2 USING ( column1 ,... ) ...
```

A NATURAL JOIN is similar to a **JOIN...USING**, only it automatically tests for equality between the values of every column that exists in both tables:

```
SELECT ... FROM table1 NATURAL JOIN table2...
```

Based on the above tables, we can write a INNER JOIN as follows:

```
sqlite> SELECT EMP_ID, NAME, DEPT FROM COMPANY INNER JOIN DEPARTMENT
        ON COMPANY.ID = DEPARTMENT.EMP_ID;
```

Above query will produce the following result:

```
EMP_ID      NAME        DEPT
----------  ----------  ----------
1           Paul        IT Billing
2           Allen       Engineerin
7           James       Finance
```

The OUTER JOIN

The OUTER JOIN is an extension of the INNER JOIN. Though SQL standard defines three types of OUTER JOINs: LEFT, RIGHT and FULL but SQLite only supports the **LEFT OUTER JOIN**.

The OUTER JOINs have a condition that is identical to INNER JOINs, expressed using an ON, USING or NATURAL keyword. The initial results table is calculated the same way. Once the primary JOIN is calculated, an OUTER join will take any unjoined rows from one or both tables, pad them out with NULLs, and append them to the resulting table.

Following is the syntax of LEFT OUTER JOIN:

```
SELECT ... FROM table1 LEFT OUTER JOIN table2 ON conditional_expression ...
```

To avoid redundancy and keep the phrasing shorter, OUTER JOIN conditions can be declared with a USING expression. This expression specifies a list of one or more columns:

```
SELECT ... FROM table1 LEFT OUTER JOIN table2 USING ( column1 ,... ) ...
```

Based on the above tables, we can write a inner join as follows:

```
sqlite> SELECT EMP_ID, NAME, DEPT FROM COMPANY LEFT OUTER JOIN DEPARTMENT
        ON COMPANY.ID = DEPARTMENT.EMP_ID;
```

Above query will produce the following result:

```
EMP_ID      NAME        DEPT
----------  ----------  ----------
1           Paul        IT Billing
2           Allen       Engineerin
```

```
            Teddy
            Mark
            David
            Kim
7           James        Finance
```

CHAPTER

29

SQLite UNIONS Clause

The SQLite **UNION** clause/operator is used to combine the results of two or more SELECT statements without returning any duplicate rows.

To use UNION, each SELECT must have the same number of columns selected, the same number of column expressions, the same data type, and have them in the same order, but they do not have to be the same length.

Syntax:

The basic syntax of **UNION** is as follows:

```
SELECT column1 [, column2 ]
FROM table1 [, table2 ]
[WHERE condition]

UNION

SELECT column1 [, column2 ]
FROM table1 [, table2 ]
[WHERE condition]
```

Here given condition could be any given expression based on your requirement.

Example:

The tables COMPANY and DEPARTMENT are shown here:

COMPANY TABLE

This is the file to create COMPANY table and to populate it with 7 records.

-- Just copy and past them on sqlite> prompt.

```
DROP TABLE COMPANY;
CREATE TABLE COMPANY(
   ID INT PRIMARY KEY     NOT NULL,
   NAME           TEXT    NOT NULL,
   AGE            INT     NOT NULL,
   ADDRESS        CHAR(50),
   SALARY         REAL
);
```

```
INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (1, 'Paul', 32, 'California', 20000.00 );

INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (2, 'Allen', 25, 'Texas', 15000.00 );

INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (3, 'Teddy', 23, 'Norway', 20000.00 );

INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (4, 'Mark', 25, 'Rich-Mond ', 65000.00 );

INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (5, 'David', 27, 'Texas', 85000.00 );

INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (6, 'Kim', 22, 'South-Hall', 45000.00 );

INSERT INTO COMPANY VALUES (7, 'James', 24, 'Houston', 10000.00 );
```

DEPARTMENT TABLE

-- This is the file to create DEPARTMENT table and to populate it with 7 records.

-- Just copy and past them on sqlite> prompt.

```
DROP TABLE DEPARTMENT;
CREATE TABLE DEPARTMENT(
   ID INT PRIMARY KEY      NOT NULL,
   DEPT           CHAR(50) NOT NULL,
   EMP_ID         INT      NOT NULL
);
```

```
INSERT INTO DEPARTMENT(ID,DEPT,EMP_ID)
VALUES (1, 'IT Billing', 1);
INSERT INTO DEPARTMENT(ID,DEPT,EMP_ID)
VALUES (2, 'Engineering', 2);
INSERT INTO DEPARTMENT(ID,DEPT,EMP_ID)
VALUES (3, 'Finance',7);
INSERT INTO DEPARTMENT(ID,DEPT,EMP_ID)
VALUES (4, 'Engineering',3);
INSERT INTO DEPARTMENT(ID,DEPT,EMP_ID)
VALUES (5, 'Finance', 4);
INSERT INTO DEPARTMENT(ID,DEPT,EMP_ID)
VALUES (6, 'Engineering', 5);
INSERT INTO DEPARTMENT(ID,DEPT,EMP_ID)
VALUES (7, 'Finance', 6);
```

Consider following two tables, (a) COMPANY table is as follows:

```
sqlite> select * from COMPANY;
ID          NAME                  AGE         ADDRESS     SALARY
----------  --------------------  ----------  ----------  ----------
1           Paul                  32          California  20000.0
2           Allen                 25          Texas       15000.0
3           Teddy                 23          Norway      20000.0
4           Mark                  25          Rich-Mond   65000.0
5           David                 27          Texas       85000.0
```

```
6           Kim                   22          South-Hall  45000.0
7           James                 24          Houston     10000.0
```

(b) Another table is DEPARTMENT as follows:

```
ID          DEPT                  EMP_ID
----------  --------------------  ----------
1           IT Billing            1
2           Engineering           2
3           Finance               7
4           Engineering           3
5           Finance               4
6           Engineering           5
7           Finance               6
```

Now, let us join these two tables using SELECT statement along with UNION clause as follows:

```
sqlite> SELECT EMP_ID, NAME, DEPT FROM COMPANY INNER JOIN DEPARTMENT
        ON COMPANY.ID = DEPARTMENT.EMP_ID
   UNION
     SELECT EMP_ID, NAME, DEPT FROM COMPANY LEFT OUTER JOIN DEPARTMENT
        ON COMPANY.ID = DEPARTMENT.EMP_ID;
```

This would produce the following result:

```
EMP_ID      NAME                  DEPT
----------  --------------------  ----------
1           Paul                  IT Billing
2           Allen                 Engineerin
3           Teddy                 Engineerin
4           Mark                  Finance
5           David                 Engineerin
6           Kim                   Finance
7           James                 Finance
```

The UNION ALL Clause:

The UNION ALL operator is used to combine the results of two SELECT statements including duplicate rows.

The same rules that apply to UNION apply to the UNION ALL operator as well.

Syntax:

The basic syntax of **UNION ALL** is as follows:

```
SELECT column1 [, column2 ]
FROM table1 [, table2 ]
[WHERE condition]

UNION ALL

SELECT column1 [, column2 ]
FROM table1 [, table2 ]
[WHERE condition]
```

Here given condition could be any given expression based on your requirement.

Example:

Now, let us join above-mentioned two tables in our SELECT statement as follows:

```
sqlite> SELECT EMP_ID, NAME, DEPT FROM COMPANY INNER JOIN DEPARTMENT
        ON COMPANY.ID = DEPARTMENT.EMP_ID
   UNION ALL
     SELECT EMP_ID, NAME, DEPT FROM COMPANY LEFT OUTER JOIN DEPARTMENT
        ON COMPANY.ID = DEPARTMENT.EMP_ID;
```

This would produce the following result:

```
EMP_ID      NAME                  DEPT
----------  --------------------  ----------
1           Paul                  IT Billing
2           Allen                 Engineerin
3           Teddy                 Engineerin
4           Mark                  Finance
5           David                 Engineerin
6           Kim                   Finance
7           James                 Finance
1           Paul                  IT Billing
2           Allen                 Engineerin
3           Teddy                 Engineerin
4           Mark                  Finance
5           David                 Engineerin
6           Kim                   Finance
7           James                 Finance
```

CHAPTER

30

SQLite NULL Values

The SQLite **NULL** is the term used to represent a missing value. A NULL value in a table is a value in a field that appears to be blank.

A field with a NULL value is a field with no value. It is very important to understand that a NULL value is different than a zero value or a field that contains spaces.

Syntax:

The basic syntax of using **NULL** while creating a table:

```
SQLite> CREATE TABLE COMPANY(
   ID INT PRIMARY KEY     NOT NULL,
   NAME           TEXT    NOT NULL,
   AGE            INT     NOT NULL,
   ADDRESS        CHAR(50),
   SALARY         REAL
);
```

Here, **NOT NULL** signifies that column should always accept an explicit value of the given data type. There are two columns where we did not use NOT NULL which means these columns could be NULL.

A field with a NULL value is one that has been left blank during record creation.

Example:

The NULL value can cause problems when selecting data, however, because when comparing an unknown value to any other value, the result is always unknown and not included in the final results. Consider the following table, COMPANY having the following records:

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

Let us use UPDATE statement to set few nullable values as NULL as follows:

```
sqlite> UPDATE COMPANY SET ADDRESS = NULL, SALARY = NULL where ID IN(6,7);
```

Now, COMPANY table should have the following records:

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22
7           James       24
```

Next, let us see the usage of **IS NOT NULL** operator to list down all the records where SALARY is not NULL:

```
sqlite> SELECT  ID, NAME, AGE, ADDRESS, SALARY
        FROM COMPANY
        WHERE SALARY IS NOT NULL;
```

Above SQLite statement will produce the following result:

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

Following is the usage of **IS NULL** operator, which will list down all the records where SALARY is NULL:

```
sqlite> SELECT  ID, NAME, AGE, ADDRESS, SALARY
        FROM COMPANY
        WHERE SALARY IS NULL;
```

Above SQLite statement will produce the following result:

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
6           Kim         22
7           James       24
```

CHAPTER

31

SQLite ALIAS Syntax

You can rename a table or a column temporarily by giving another name, which is known as **ALIAS**. The use of table aliases means to rename a table in a particular SQLite statement. Renaming is a temporary change and the actual table name does not change in the database.

The column aliases are used to rename a table's columns for the purpose of a particular SQLite query.

Syntax:

The basic syntax of **table** alias is as follows:

```
SELECT column1, column2....
FROM table_name AS alias_name
WHERE [condition];
```

The basic syntax of **column** alias is as follows:

```
SELECT column_name AS alias_name
FROM table_name
WHERE [condition];
```

Example:

Consider the following two tables, (a) COMPANY table is as follows:

```
sqlite> select * from COMPANY;
ID          NAME                  AGE         ADDRESS     SALARY
----------  --------------------  ----------  ----------  ----------
1           Paul                  32          California  20000.0
2           Allen                 25          Texas       15000.0
3           Teddy                 23          Norway      20000.0
4           Mark                  25          Rich-Mond   65000.0
5           David                 27          Texas       85000.0
6           Kim                   22          South-Hall  45000.0
7           James                 24          Houston     10000.0
```

(b) Another table is DEPARTMENT as follows:

```
ID          DEPT                  EMP_ID
----------  --------------------  ----------
```

```
1           IT Billing           1
2           Engineering          2
3           Finance              7
4           Engineering          3
5           Finance              4
6           Engineering          5
7           Finance              6
```

Now, following is the usage of **TABLE ALIAS** where we use C and D as aliases for COMPANY and DEPARTMENT tables respectively:

```
sqlite> SELECT C.ID, C.NAME, C.AGE, D.DEPT
        FROM COMPANY AS C, DEPARTMENT AS D
        WHERE  C.ID = D.EMP_ID;
```

Above SQLite statement will produce the following result:

```
ID          NAME        AGE         DEPT
----------  ----------  ----------  ----------
1           Paul        32          IT Billing
2           Allen       25          Engineerin
3           Teddy       23          Engineerin
4           Mark        25          Finance
5           David       27          Engineerin
6           Kim         22          Finance
7           James       24          Finance
```

Let us see an example for the usage of **COLUMN ALIAS** where COMPANY_ID is an alias of ID column and COMPANY_NAME is an alias of name column:

```
sqlite> SELECT C.ID AS COMPANY_ID, C.NAME AS COMPANY_NAME, C.AGE, D.DEPT
        FROM COMPANY AS C, DEPARTMENT AS D
        WHERE  C.ID = D.EMP_ID;
```

Above SQLite statement will produce the following result:

```
COMPANY_ID  COMPANY_NAME  AGE         DEPT
----------  ------------  ----------  ----------
1           Paul          32          IT Billing
2           Allen         25          Engineerin
3           Teddy         23          Engineerin
4           Mark          25          Finance
5           David         27          Engineerin
6           Kim           22          Finance
7           James         24          Finance
```

CHAPTER

32

SQLite Triggers

SQLite **Triggers** are database callback functions, which are automatically performed/invoked when a specified database event occurs. Following are the important points about SQLite triggers:

- SQLite trigger may be specified to fire whenever a DELETE, INSERT or UPDATE of a particular database table occurs or whenever an UPDATE occurs on on one or more specified columns of a table.
- At this time, SQLite supports only FOR EACH ROW triggers, not FOR EACH STATEMENT triggers. Hence, explicitly specifying FOR EACH ROW is optional.
- Both the WHEN clause and the trigger actions may access elements of the row being inserted, deleted or updated using references of the form **NEW.column-name** and **OLD.column-name**, where column-name is the name of a column from the table that the trigger is associated with.
- If a WHEN clause is supplied, the SQL statements specified are only executed for rows for which the WHEN clause is true. If no WHEN clause is supplied, the SQL statements are executed for all rows.
- The BEFORE or AFTER keyword determines when the trigger actions will be executed relative to the insertion, modification or removal of the associated row.
- Triggers are automatically dropped when the table that they are associated with is dropped.
- The table to be modified must exist in the same database as the table or view to which the trigger is attached and one must use just **tablename** not **database.tablename**.
- A special SQL function RAISE() may be used within a trigger-program to raise an exception.

Syntax:

The basic syntax of creating a **trigger** is as follows:

```
CREATE  TRIGGER trigger_name [BEFORE|AFTER] event_name
ON table_name
BEGIN
 -- Trigger logic goes here....
END;
```

Here, **event_name** could be INSERT, DELETE, and UPDATE database operation on the mentioned table **table_name**. You can optionally specify FOR EACH ROW after table name.

Following is the syntax of creating a trigger on an UPDATE operation on one or more specified columns of a table as follows:

```
CREATE  TRIGGER trigger_name [BEFORE|AFTER] UPDATE OF column_name
ON table_name
BEGIN
 -- Trigger logic goes here....
END;
```

Example

Let us consider a case where we want to keep audit trial for every record being inserted in COMPANY table, which we create newly as follows (Drop COMPANY table if you already have it):

```
sqlite> CREATE TABLE COMPANY(
   ID INT PRIMARY KEY     NOT NULL,
   NAME           TEXT    NOT NULL,
   AGE            INT     NOT NULL,
   ADDRESS        CHAR(50),
   SALARY         REAL
);
```

To keep audit trial, we will create a new table called AUDIT where log messages will be inserted whenever there is an entry in COMPANY table for a new record:

```
sqlite> CREATE TABLE AUDIT(
    EMP_ID INT NOT NULL,
    ENTRY_DATE TEXT NOT NULL
);
```

Here, ID is the AUDIT record ID, and EMP_ID is the ID which will come from COMPANY table and DATE will keep timestamp when the record will be created in COMPANY table. So now let's create a trigger on COMPANY table as follows:

```
sqlite> CREATE TRIGGER audit_log AFTER INSERT
ON COMPANY
BEGIN
   INSERT INTO AUDIT(EMP_ID, ENTRY_DATE) VALUES (new.ID, datetime('now'));
END;
```

Now, we will start actual work, let's start inserting record in COMPANY table which should result in creating an audit log record in AUDIT table. So let's create one record in COMPANY table as follows:

```
sqlite> INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (1, 'Paul', 32, 'California', 20000.00 );
```

This will create one record in COMPANY table, which is as follows:

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
```

Same time, one record will be create in AUDIT table. This record is the result of a trigger, which we have created on INSERT operation on COMPANY table. Similar way, you can create your triggers on UPDATE and DELETE operations based on your requirements.

```
EMP_ID      ENTRY_DATE
----------  -------------------
1           2013-04-05 06:26:00
```

Listing TRIGGERS

You can list down all the triggers from **sqlite_master** table as follows:

```
sqlite> SELECT name FROM sqlite_master
WHERE type = 'trigger';
```

Above SQLite statement will list down only one entry as follows:

```
name
----------
audit_log
```

If you want to list down triggers on a particular table, then use AND clause with table name as follows:

```
sqlite> SELECT name FROM sqlite_master
WHERE type = 'trigger' AND tbl_name = 'COMPANY';
```

Above SQLite statement will also list down only one entry as follows:

```
name
----------
audit_log
```

Dropping TRIGGERS

Following is the DROP command, which can be used to drop an existing trigger:

```
sqlite> DROP TRIGGER trigger_name;
```

CHAPTER

33

SQLite Indexes

Indexes are special lookup tables that the database search engine can use to speed up data retrieval. Simply put, an index is a pointer to data in a table. An index in a database is very similar to an index in the back of a book.

For example, if you want to reference all pages in a book that discuss a certain topic, you first refer to the index, which lists all topics alphabetically and are then referred to one or more specific page numbers.

An index helps speed up SELECT queries and WHERE clauses, but it slows down data input, with UPDATE and INSERT statements. Indexes can be created or dropped with no effect on the data.

Creating an index involves the CREATE INDEX statement, which allows you to name the index, to specify the table and which column or columns to index, and to indicate whether the index is in ascending or descending order.

Indexes can also be unique, similar to the UNIQUE constraint, in that the index prevents duplicate entries in the column or combination of columns on which there's an index.

The CREATE INDEX Command:

The basic syntax of **CREATE INDEX** is as follows:

```
CREATE INDEX index_name ON table_name;
```

Single-Column Indexes:

A single-column index is one that is created based on only one table column. The basic syntax is as follows:

```
CREATE INDEX index_name
ON table_name (column_name);
```

Unique Indexes:

Unique indexes are used not only for performance, but also for data integrity. A unique index does not allow any duplicate values to be inserted into the table. The basic syntax is as follows:

```
CREATE INDEX index_name
```

```
on table_name (column_name);
```

Composite Indexes:

A composite index is an index on two or more columns of a table. The basic syntax is as follows:

```
CREATE INDEX index_name
on table_name (column1, column2);
```

Whether to create a single-column index or a composite index, take into consideration the column(s) that you may use very frequently in a query's WHERE clause as filter conditions.

Should there be only one column used, a single-column index should be the choice. Should there be two or more columns that are frequently used in the WHERE clause as filters, the composite index would be the best choice.

Implicit Indexes:

Implicit indexes are indexes that are automatically created by the database server when an object is created. Indexes are automatically created for primary key constraints and unique constraints.

Example

Following is an example where we will create an index on COMPANY table for salary column:

```
sqlite> CREATE INDEX salary_index ON COMPANY (salary);
```

Now, let's list down all the indices available on COMPANY table using **.indices** command as follows:

```
sqlite> .indices COMPANY
```

This will produce the following result, where sqlite_autoindex_COMPANY_1 is an implicit index, which got created when table itself was created.

```
salary_index
sqlite_autoindex_COMPANY_1
```

You can list down all the indexes database wide as follows:

```
sqlite> SELECT * FROM sqlite_master WHERE type = 'index';
```

The DROP INDEX Command:

An index can be dropped using SQLite **DROP** command. Care should be taken when dropping an index because performance may be slowed or improved.

The basic syntax is as follows:

```
DROP INDEX index_name;
```

You can use following statement to delete previously created index:

```
sqlite> DROP INDEX salary_index;
```

When should indexes be avoided?

Although indexes are intended to enhance a database's performance, there are times when they should be avoided. The following guidelines indicate when the use of an index should be reconsidered:

- Indexes should not be used on small tables.
- Tables that have frequent, large batch update or insert operations.
- Indexes should not be used on columns that contain a high number of NULL values.
- Columns that are frequently manipulated should not be indexed.

CHAPTER

34

SQLite Indexed By

The "INDEXED BY index-name" clause specifies that the named index must be used in order to look up values on the preceding table.

If index-name does not exist or cannot be used for the query, then the preparation of the SQLite statement fails.

The "NOT INDEXED" clause specifies that no index shall be used when accessing the preceding table, including implied indices create by UNIQUE and PRIMARY KEY constraints.

However, the INTEGER PRIMARY KEY can still be used to look up entries even when "NOT INDEXED" is specified.

Syntax

Following is the syntax for INDEXED BY clause and it can be used with DELETE, UPDATE or SELECT statement:

```
SELECT|DELETE|UPDATE column1, column2...
INDEXED BY (index_name)
table_name
WHERE (CONDITION);
```

Example

COMPANY Table:

```
-- This is the file to create COMPANY table and to populate it with 7 records.
-- Just copy and past them on sqlite> prompt.
DROP TABLE COMPANY;
CREATE TABLE COMPANY(
   ID INT PRIMARY KEY     NOT NULL,
   NAME           TEXT    NOT NULL,
   AGE            INT     NOT NULL,
   ADDRESS        CHAR(50),
   SALARY         REAL
);
INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (1, 'Paul', 32, 'California', 20000.00 );

INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (2, 'Allen', 25, 'Texas', 15000.00 );
```

```
INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (3, 'Teddy', 23, 'Norway', 20000.00 );

INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (4, 'Mark', 25, 'Rich-Mond ', 65000.00 );

INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (5, 'David', 27, 'Texas', 85000.00 );

INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (6, 'Kim', 22, 'South-Hall', 45000.00 );

INSERT INTO COMPANY VALUES (7, 'James', 24, 'Houston', 10000.00 );
```

Consider above table COMPANY we will create an index and use it for performing INDEXED BY operation.

```
sqlite> CREATE INDEX salary_index ON COMPANY(salary);
sqlite>
```

Now selecting the data from table COMPANY you can use INDEXED BY clause as follows:

```
sqlite> SELECT * FROM COMPANY INDEXED BY salary_index WHERE salary > 5000;
```

Kindly note that though SQLite specification talks about the above-mentioned Syntax for**INDEXED BY** clause but I tried all the way to make INDEXED BY work on my installation but it did not work. If you found a solution kindly share it at webmaster@tutorialspoint.com.

CHAPTER

35

SQLite Alter Command

The SQLite **ALTER TABLE** command modifies an existing table without performing a full dump and reload of the data. You can rename a table using ALTER TABLE statement and additional columns can be added in an existing table using ALTER TABLE statement.

There is no other operation supported by ALTER TABLE command in SQLite except renaming a table and adding a column in existing table.

Syntax:

The basic syntax of **ALTER TABLE** to RENAME an existing table is as follows:

```
ALTER TABLE database_name.table_name RENAME TO new_table_name;
```

The basic syntax of **ALTER TABLE** to add a new column in an existing table is as follows:

```
ALTER TABLE database_name.table_name ADD COLUMN column_def...;
```

Example:

Consider our COMPANY table has the following records:

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

Now, let's try to rename this table using ALTER TABLE statement as follows:

```
sqlite> ALTER TABLE COMPANY RENAME TO OLD_COMPANY;
```

Above SQLite statement will rename COMPANY table to OLD_COMPANY. Now, let's try to add a new column in OLD_COMPANY table as follows:

```
sqlite> ALTER TABLE OLD_COMPANY ADD COLUMN SEX char(1);
```

Now, COMPANY table is changed and following would be output from SELECT statement:

```
ID          NAME        AGE         ADDRESS     SALARY      SEX
----------  ----------  ----------  ----------  ----------  ---
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

It should be noted that newly added column is filled with NULL values.

CHAPTER 36

SQLite Truncate Table

Unfortunately, we do not have TRUNCATE TABLE command in SQLite but you can use SQLite **DELETE** command to delete complete data from an existing table, though it is recommended to use DROP TABLE command to drop complete table and re-create it once again.

Syntax:

The basic syntax of DELETE command is as follows:

```
sqlite> DELETE FROM table_name;
```

The basic syntax of DROP TABLE is as follows:

```
sqlite> DROP TABLE table_name;
```

If you are using DELETE TABLE command to delete all the records, it is recommended to use**VACUUM** command to clear unused space.

Example:

Consider COMPANY table is having the following records:

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

Following is the example to truncate the above table:

```
SQLite> DELETE FROM COMPANY;
SQLite> VACUUM;
```

Now, COMPANY table is truncated completely and nothing would be output from SELECT statement:

CHAPTER

37

SQLite Views

A view is nothing more than a SQLite statement that is stored in the database with an associated name.

A view is actually a composition of a table in the form of a predefined SQLite query.

A view can contain all rows of a table or selected rows from one or more tables. A view can be created from one or many tables which depends on the written SQLite query to create a view.

Views which are kind of virtual tables, allow users to do the following:

- Structure data in a way that users or classes of users find natural or intuitive.
- Restrict access to the data such that a user can only see limited data instead of complete table.
- Summarize data from various tables which can be used to generate reports.

SQLite views are read-only and so you may not execute a DELETE, INSERT or UPDATE statement on a view. But you can create a trigger on a view that fires on an attempt to DELETE, INSERT or UPDATE a view and do what you need in the body of the trigger.

Creating Views:

The SQLite views are created using the **CREATE VIEW** statement. The SQLIte views can be created from a single table, multiple tables, or another view.

The basic CREATE VIEW syntax is as follows:

```
CREATE [TEMP | TEMPORARY] VIEW view_name AS
SELECT column1, column2.....
FROM table_name
WHERE [condition];
```

You can include multiple tables in your SELECT statement in very similar way as you use them in normal SQL SELECT query. If the optional TEMP or TEMPORARY keyword is present, the view will be created in the temp database.

Example:

Consider COMPANY table is having the following records:

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
```

```
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

Now, following is an example to create a view from COMPANY table. This view would be used to have only few columns from COMPANY table:

```
sqlite> CREATE VIEW COMPANY_VIEW AS
SELECT ID, NAME, AGE
FROM  COMPANY;
```

Now, you can query COMPANY_VIEW in similar way as you query an actual table. Following is the example:

```
sqlite> SELECT * FROM COMPANY_VIEW;
```

This would produce the following result:

```
ID          NAME        AGE
----------  ----------  ----------
1           Paul        32
2           Allen       25
3           Teddy       23
4           Mark        25
5           David       27
6           Kim         22
7           James       24
```

Dropping Views:

To drop a view, simply use the DROP VIEW statement with the **view_name**. The basic DROP VIEW syntax is as follows:

```
sqlite> DROP VIEW view_name;
```

Following command will delete COMPANY_VIEW view, which we created in the last section:

```
sqlite> DROP VIEW COMPANY_VIEW;
```

CHAPTER

38

SQLite TRANSACTIONS

A transaction is a unit of work that is performed against a database. Transactions are units or sequences of work accomplished in a logical order, whether in a manual fashion by a user or automatically by some sort of a database program.

A transaction is the propagation of one or more changes to the database. For example, if you are creating a record or updating a record or deleting a record from the table. then you are performing transaction on the table. It is important to control transactions to ensure data integrity and to handle database errors.

Practically, you will club many SQLite queries into a group and you will execute all of them together as part of a transaction.

Properties of Transactions:

Transactions have the following four standard properties, usually referred to by the acronym ACID:

- **Atomicity:** ensures that all operations within the work unit are completed successfully; otherwise, the transaction is aborted at the point of failure and previous operations are rolled back to their former state.
- **Consistency:** ensures that the database properly changes states upon a successfully committed transaction.
- **Isolation:** enables transactions to operate independently of and transparent to each other.
- **Durability:** ensures that the result or effect of a committed transaction persists in case of a system failure.

Transaction Control:

There are the following commands used to control transactions:

- **BEGIN TRANSACTION:** to start a transaction.
- **COMMIT:** to save the changes, alternatively you can use **END TRANSACTION** command.
- **ROLLBACK:** to rollback the changes.

Transactional control commands are only used with the DML commands INSERT, UPDATE and DELETE. They can not be used while creating tables or dropping them because these operations are automatically committed in the database.

The BEGIN TRANSACTION Command:

Transactions can be started using BEGIN TRANSACTION or simply BEGIN command. Such transactions usually persist until the next COMMIT or ROLLBACK command encountered. But a transaction will also ROLLBACK if the database is closed or if an error occurs. Following is the simple syntax to start a transaction:

```
BEGIN;

or

BEGIN TRANSACTION;
```

The COMMIT Command:

The COMMIT command is the transactional command used to save changes invoked by a transaction to the database.

The COMMIT command saves all transactions to the database since the last COMMIT or ROLLBACK command.

The syntax for COMMIT command is as follows:

```
COMMIT;

or

END TRANSACTION;
```

The ROLLBACK Command:

The ROLLBACK command is the transactional command used to undo transactions that have not already been saved to the database.

The ROLLBACK command can only be used to undo transactions since the last COMMIT or ROLLBACK command was issued.

The syntax for ROLLBACK command is as follows:

```
ROLLBACK;
```

Example:

Consider COMPANY table is having the following records:

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

Now, let's start a transaction and delete records from the table having age = 25 and finally we use ROLLBACK command to undo all the changes.

```
sqlite> BEGIN;
sqlite> DELETE FROM COMPANY WHERE AGE = 25;
sqlite> ROLLBACK;
```

If you will check, COMPANY table is still having the following records:

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

Now, let's start another transaction and delete records from the table having age = 25 and finally we use COMMIT command to commit all the changes.

```
sqlite> BEGIN;
sqlite> DELETE FROM COMPANY WHERE AGE = 25;
sqlite> COMMIT;
```

If you will check, COMPANY table is still having the following records:

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
3           Teddy       23          Norway      20000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

CHAPTER

39

SQLite Sub Queries

A Subquery or Inner query or Nested query is a query within another SQLite query and embedded within the WHERE clause.

A subquery is used to return data that will be used in the main query as a condition to further restrict the data to be retrieved.

Subqueries can be used with the SELECT, INSERT, UPDATE and DELETE statements along with the operators like =, <, >, >=, <=, IN, BETWEEN etc.

There are a few rules that subqueries must follow:

- Subqueries must be enclosed within parentheses.
- A subquery can have only one column in the SELECT clause, unless multiple columns are in the main query for the subquery to compare its selected columns.
- An ORDER BY cannot be used in a subquery, although the main query can use an ORDER BY. The GROUP BY can be used to perform the same function as the ORDER BY in a subquery.
- Subqueries that return more than one row can only be used with multiple value operators, such as the IN operator.
- The BETWEEN operator cannot be used with a subquery; however, the BETWEEN can be used within the subquery.

Subqueries with the SELECT Statement:

Subqueries are most frequently used with the SELECT statement. The basic syntax is as follows:

```
SELECT column_name [, column_name ]
FROM   table1 [, table2 ]
WHERE  column_name OPERATOR
      (SELECT column_name [, column_name ]
      FROM table1 [, table2 ]
      [WHERE])
```

Example:

Consider COMPANY table is having the following records:

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

Now, let us check following sub-query with SELECT statement:

```
sqlite> SELECT *
     FROM COMPANY
     WHERE ID IN (SELECT ID
                  FROM COMPANY
                  WHERE SALARY > 45000) ;
```

This would produce the following result:

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

Subqueries with the INSERT Statement:

Subqueries also can be used with INSERT statements. The INSERT statement uses the data returned from the subquery to insert into another table. The selected data in the subquery can be modified with any of the character, date or number functions.

The basic syntax is as follows:

```
INSERT INTO table_name [ (column1 [, column2 ]) ]
           SELECT [ *|column1 [, column2 ]
           FROM table1 [, table2 ]
           [ WHERE VALUE OPERATOR ]
```

Example:

Consider a table COMPANY_BKP with similar structure as COMPANY table and can be created using same CREATE TABLE using COMPANY_BKP as table name. Now to copy complete COMPANY table into COMPANY_BKP, following is the syntax:

```
sqlite> INSERT INTO COMPANY_BKP
     SELECT * FROM COMPANY
     WHERE ID IN (SELECT ID
                  FROM COMPANY) ;
```

Subqueries with the UPDATE Statement:

The subquery can be used in conjunction with the UPDATE statement. Either single or multiple columns in a table can be updated when using a subquery with the UPDATE statement.

The basic syntax is as follows:

```
UPDATE table
SET column_name = new_value
[ WHERE OPERATOR [ VALUE ]
   (SELECT COLUMN_NAME
   FROM TABLE_NAME)
   [ WHERE) ]
```

Example:

Assuming, we have COMPANY_BKP table available which is backup of COMPANY table.

Following example updates SALARY by 0.50 times in COMPANY table for all the customers, whose AGE is greater than or equal to 27:

```
sqlite> UPDATE COMPANY
     SET SALARY = SALARY * 0.50
     WHERE AGE IN (SELECT AGE FROM COMPANY_BKP
                   WHERE AGE >= 27 );
```

This would impact two rows and finally COMPANY table would have the following records:

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  10000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       42500.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

Subqueries with the DELETE Statement:

The subquery can be used in conjunction with the DELETE statement like with any other statements mentioned above.

The basic syntax is as follows:

```
DELETE FROM TABLE_NAME
[ WHERE OPERATOR [ VALUE ]
   (SELECT COLUMN_NAME
   FROM TABLE_NAME)
   [ WHERE) ]
```

Example:

Assuming, we have COMPANY_BKP table available which is backup of COMPANY table.

Following example deletes records from COMPANY table for all the customers whose AGE is greater than or equal to 27:

```
sqlite> DELETE FROM COMPANY
     WHERE AGE IN (SELECT AGE FROM COMPANY_BKP
                   WHERE AGE > 27 );
```

This would impact two rows and finally COMPANY table would have the following records:

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       42500.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

CHAPTER

40

SQLite AUTOINCREMENT

SQLite **AUTOINCREMENT** is a keyword used for auto incrementing a value of a field in the table. We can auto increment a field value by using **AUTOINCREMENT** keyword when creating a table with specific column name to auto incrementing it.
The keyword **AUTOINCREMENT** can be used with INTEGER field only.

Syntax:

The basic usage of **AUTOINCREMENT** keyword is as follows:

```
CREATE TABLE table_name(
   column1 INTEGER AUTOINCREMENT,
   column2 datatype,
   column3 datatype,
   .....
   columnN datatype,
);
```

Example:

Consider COMPANY table to be created as follows:

```
sqlite> CREATE TABLE COMPANY(
   ID INTEGER PRIMARY KEY   AUTOINCREMENT,
   NAME           TEXT      NOT NULL,
   AGE            INT       NOT NULL,
   ADDRESS        CHAR(50),
   SALARY         REAL
);
```

Now, insert following records into table COMPANY:

```
INSERT INTO COMPANY (NAME,AGE,ADDRESS,SALARY)
VALUES ( 'Paul', 32, 'California', 20000.00 );

INSERT INTO COMPANY (NAME,AGE,ADDRESS,SALARY)
VALUES ('Allen', 25, 'Texas', 15000.00 );

INSERT INTO COMPANY (NAME,AGE,ADDRESS,SALARY)
VALUES ('Teddy', 23, 'Norway', 20000.00 );
```

```
INSERT INTO COMPANY (NAME,AGE,ADDRESS,SALARY)
VALUES ( 'Mark', 25, 'Rich-Mond ', 65000.00 );

INSERT INTO COMPANY (NAME,AGE,ADDRESS,SALARY)
VALUES ( 'David', 27, 'Texas', 85000.00 );


INSERT INTO COMPANY (NAME,AGE,ADDRESS,SALARY)
VALUES ( 'Kim', 22, 'South-Hall', 45000.00 );

INSERT INTO COMPANY (NAME,AGE,ADDRESS,SALARY)
VALUES ( 'James', 24, 'Houston', 10000.00 );
```

This will insert 7 tuples into the table COMPANY and COMPANY will have the following records:

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

CHAPTER

41

SQLite Injection

If you take user input through a webpage and insert it into a SQLite database, there's a chance that you have left yourself wide open for a security issue known as SQL Injection. This lesson will teach you how to help prevent this from happening and help you secure your scripts and SQLite statements.

Injection usually occurs when you ask a user for input, like their name, and instead of a name they give you a SQLite statement that you will unknowingly run on your database.

Never trust user provided data, process this data only after validation; as a rule, this is done by pattern matching. In the example below, the username is restricted to alphanumerical chars plus underscore and to a length between 8 and 20 chars - modify these rules as needed.

```
if (preg_match("/^\w{8,20}$/", $_GET['username'], $matches)){
   $db = new SQLiteDatabase('filename');
   $result = @$db->query("SELECT * FROM users WHERE username=$matches[0]");
}else{
   echo "username not accepted";
}
```

To demonstrate the problem, consider this excerpt:

```
$name = "Qadir'; DELETE FROM users;";
@$db->query("SELECT * FROM users WHERE username='{$name}'");
```

The function call is supposed to retrieve a record from the users table where the name column matches the name specified by the user. Under normal circumstances, **$name** would only contain alphanumeric characters and perhaps spaces, such as the string ilia. But here, by appending an entirely new query to $name, the call to the database turns into disaster: the injected DELETE query removes all records from users.

There are database's interfaces which do not permit query stacking or executing multiple queries in a single function call. If you try to stack queries, the call fails but SQLite and PostgreSQL, happily perform stacked queries, executing all of the queries provided in one string and creating a serious security problem.

Preventing SQL Injection:

You can handle all escape characters smartly in scripting languages like PERL and PHP. Programming language PHP provides the function **string sqlite_escape_string()** to escape input characters that are special to SQLite.

```
if (get_magic_quotes_gpc())
{
  $name = sqlite_escape_string($name);
```

```
}
$result = @$db->query("SELECT * FROM users WHERE username='{$name}'");
```

Although the encoding makes it safe to insert the data, it will render simple text comparisons and **LIKE**clauses in your queries unusable for the columns that contain the binary data.

Keep a note that addslashes() should NOT be used to quote your strings for SQLite queries; it will lead to strange results when retrieving your data.

CHAPTER

42

SQLite Explain

An SQLite statement can be preceded by the keyword "EXPLAIN" or by the phrase "EXPLAIN QUERY PLAN" used for describing the details of table.

Either modification causes the SQLite statement to behave as a query and to return information about how the SQLite statement would have operated if the EXPLAIN keyword or phrase had been omitted.

- The output from EXPLAIN and EXPLAIN QUERY PLAN is intended for interactive analysis and troubleshooting only.
- The details of the output format are subject to change from one release of SQLite to the next.
- Applications should not use EXPLAIN or EXPLAIN QUERY PLAN since their exact behavior is variable and only partially documented.

Syntax:

Syntax for **EXPLAIN** is as follows:

```
EXPLAIN [SQLite Query]
```

Syntax for **EXPLAIN QUERY PLAN** is as follows:

```
EXPLAIN  QUERY PLAN [SQLite Query]
```

Example:

This is the file to create COMPANY table and to populate it with 7 records.

-- Just copy and past them on sqlite> prompt.

```
DROP TABLE COMPANY;
CREATE TABLE COMPANY(
   ID INT PRIMARY KEY     NOT NULL,
   NAME           TEXT    NOT NULL,
   AGE            INT     NOT NULL,
   ADDRESS        CHAR(50),
   SALARY         REAL
);
INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
```

```
VALUES (1, 'Paul', 32, 'California', 20000.00 );

INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (2, 'Allen', 25, 'Texas', 15000.00 );

INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (3, 'Teddy', 23, 'Norway', 20000.00 );

INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (4, 'Mark', 25, 'Rich-Mond ', 65000.00 );

INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (5, 'David', 27, 'Texas', 85000.00 );

INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (6, 'Kim', 22, 'South-Hall', 45000.00 );

INSERT INTO COMPANY VALUES (7, 'James', 24, 'Houston', 10000.00 );
```

Consider above shown COMPANY table is having the following records:

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

Now, let us check following sub-query with SELECT statement:

```
sqlite> EXPLAIN SELECT *  FROM COMPANY  WHERE Salary >= 20000;
```

This would produce the following result:

```
addr        opcode      p1          p2          p3
----------  ----------  ----------  ----------  ----------
0           Goto        0           19
1           Integer     0           0
2           OpenRead    0           8
3           SetNumColu  0           5
4           Rewind      0           17
5           Column      0           4
6           RealAffini  0           0
7           Integer     20000       0
8           Lt          357         16          collseq(BI
9           Rowid       0           0
10          Column      0           1
11          Column      0           2
12          Column      0           3
13          Column      0           4
14          RealAffini  0           0
15          Callback    5           0
16          Next        0           5
17          Close       0           0
18          Halt        0           0
19          Transactio  0           0
```

```
20          VerifyCook  0           38
21          Goto        0           1
22          Noop        0           0
```

Now, let us check following **Explain Query Plan** with SELECT statement:

```
SQLite> EXPLAIN QUERY PLAN SELECT * FROM COMPANY WHERE Salary >= 20000;
order       from        detail
----------  ----------  -------------
0           0           TABLE COMPANY
```

CHAPTER

43

SQLite Vacuum

The VACUUM command cleans the main database by copying its contents to a temporary database file and reloading the original database file from the copy. This eliminates free pages, aligns table data to be contiguous, and otherwise cleans up the database file structure.

The VACUUM command may change the ROWID of entries in tables that do not have an explicit INTEGER PRIMARY KEY. The VACUUM command only works on the main database. It is not possible to VACUUM an attached database file.

The VACUUM command will fail if there is an active transaction. The VACUUM command is a no-op for in-memory databases. As the VACUUM command rebuilds the database file from scratch, VACUUM can also be used to modify many database-specific configuration parameters.

Manual VACUUM

Following is simple syntax to issue a VACUUM command for the whole database from command prompt:

```
$sqlite3 database_name "VACUUM;"
```

You can run VACUUM from SQLite prompt as well as follows:

```
sqlite> VACUUM;
```

You can also run VACUUM on a particular table as follows:

```
sqlite> VACUUM table_name;
```

Auto-VACCUM

SQLite Auto-VACUUM does not do the same as VACUUM rather it only moves free pages to the end of the database thereby reducing the database size. By doing so it can significantly fragment the database while VACUUM ensures defragmentation. So Auto-VACUUM just keeps the database small.

You can enable/disable SQLite auto-vacuuming by the following pragmas running at SQLite prompt:

```
sqlite> PRAGMA auto_vacuum = NONE;  -- 0 means disable auto vacuum
sqlite> PRAGMA auto_vacuum = INCREMENTAL;  -- 1 means enable incremental vacuum
sqlite> PRAGMA auto_vacuum = FULL;  -- 2 means enable full auto vacuum
```

You can run following command from command prompt to check the auto-vacuum setting:

```
$sqlite3 database_name "PRAGMA auto_vacuum;"
```

CHAPTER

44

SQLite Date & Time

SQLite supports five date and time functions as follows:

S.N.	Function	Example
1	date(timestring, modifiers...)	This returns the date in this format: YYYY-MM-DD
2	time(timestring, modifiers...)	This returns the time as HH:MM:SS
3	datetime(timestring, modifiers...)	This returns YYYY-MM-DD HH:MM:SS
4	julianday(timestring, modifiers...)	This returns the number of days since noon in Greenwich on November 24, 4714 B.C.
5	strftime(timestring, modifiers...)	This returns the date formatted according to the format string specified as the first argument formatted as per formatters explained below.

All the above five date and time functions take a time string as an argument. The time string is followed by zero or more modifiers. The strftime() function also takes a format string as its first argument. Following section will give you detail on different types of time strings and modifiers.

Time Strings:

A time string can be in any of the following formats:

S.N.	Time String	Example
1	YYYY-MM-DD	2010-12-30
2	YYYY-MM-DD HH:MM	2010-12-30 12:10
3	YYYY-MM-DD HH:MM:SS.SSS	2010-12-30 12:10:04.100
4	MM-DD-YYYY HH:MM	30-12-2010 12:10
5	HH:MM	12:10
6	YYYY-MM-DD**T**HH:MM	2010-12-30 12:10
7	HH:MM:SS	12:10:01

8	YYYYMMDD HHMMSS	20101230 121001
9	Now	2013-05-07

You can use the "T" as a literal character separating the date and the time.

Modifiers

The time string can be followed by zero or more modifiers that will alter date and/or time returned by any of the above five functions. Modifiers are applied from left to right and following modifers are available in SQLite:

- NNN days
- NNN hours
- NNN minutes
- NNN.NNNN seconds
- NNN months
- NNN years
- start of month
- start of year
- start of day
- weekday N
- unixepoch
- localtime
- utc

Formatters:

SQLite provides very handy function **strftime()** to format any date and time. You can use following substitutions to format your date and time:

Substitution	Description
%d	Day of month, 01-31
%f	Fractional seconds, SS.SSS
%H	Hour, 00-23
%j	Day of year, 001-366
%J	Julian day number, DDDD.DDDD

%m	Month, 00-12
%M	Minute, 00-59
%s	Seconds since 1970-01-01
%S	Seconds, 00-59
%w	Day of week, 0-6 (0 is Sunday)
%W	Week of year, 01-53
%Y	Year, YYYY
%%	% symbol

Examples

Let's try various examples now using SQLite prompt. Following computes the current date:

```
sqlite> SELECT date('now');
2013-05-07
```

Following computes the last day of the current month:

```
sqlite> SELECT date('now','start of month','+1 month','-1 day');
2013-05-31
```

Following computes the date and time given a UNIX timestamp 1092941466:

```
sqlite> SELECT datetime(1092941466, 'unixepoch');
2004-08-19 18:51:06
```

Following computes the date and time given a UNIX timestamp 1092941466 and compensate for your local timezone:

```
sqlite> SELECT datetime(1092941466, 'unixepoch', 'localtime');
2004-08-19 11:51:06
```

Following computes the current UNIX timestamp:

```
sqlite> SELECT datetime(1092941466, 'unixepoch', 'localtime');
1367926057
```

Following computes the number of days since the signing of the US Declaration of Independence:

```
sqlite> SELECT julianday('now') - julianday('1776-07-04');
86504.4775830326
```

Following computes the number of seconds since a particular moment in 2004:

```
sqlite> SELECT strftime('%s','now') - strftime('%s','2004-01-01 02:34:56');
295001572
```

Following computes the date of the first Tuesday in October for the current year:

```
sqlite> SELECT date('now','start of year','+9 months','weekday 2');
2013-10-01
```

Following computes the time since the UNIX epoch in seconds (like strftime('%s','now') except includes fractional part):

```
sqlite> SELECT (julianday('now') - 2440587.5)*86400.0;
1367926077.12598
```

To convert between UTC and local time values when formatting a date, use the utc or localtime modifiers as follows:

```
sqlite> SELECT time('12:00', 'localtime');
05:00:00
sqlite>  SELECT time('12:00', 'utc');
19:00:00
```

CHAPTER

45

SQLite Useful Functions

SQLite has many built-in functions for performing processing on string or numeric data. Following is the list of few useful SQLite built-in functions and all are case insensitive, which means you can use these functions either in lower-case form or in upper-case or in mixed form. For more details, you can check official documentation for SQLite:

S.N.	Function & Description
1	**SQLite COUNT Function** The SQLite COUNT aggregate function is used to count the number of rows in a database table.
2	**SQLite MAX Function** The SQLite MAX aggregate function allows us to select the highest (maximum) value for a certain column.
3	**SQLite MIN Function** The SQLite MIN aggregate function allows us to select the lowest (minimum) value for a certain column.
4	**SQLite AVG Function** The SQLite AVG aggregate function selects the average value for certain table column.
5	**SQLite SUM Function** The SQLite SUM aggregate function allows selecting the total for a numeric column.
6	**SQLite RANDOM Function** The SQLite RANDOM function returns a pseudo-random integer between -9223372036854775808 and +9223372036854775807.
7	**SQLite ABS Function** The SQLite ABS function returns the absolute value of the numeric argument.
8	**SQLite UPPER Function** The SQLite UPPER function converts a string into upper-case letters.
9	**SQLite LOWER Function** The SQLite LOWER function converts a string into lower-case letters.
10	**SQLite LENGTH Function** The SQLite LENGTH function returns the length of a string.
11	**SQLite sqlite_version Function** The SQLite sqlite_version function returns the version of the SQLite library.

Before we start giving examples on the above mentioned functions, consider COMPANY table is having the following records:

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

SQLite COUNT Function

The SQLite COUNT aggregate function is used to count the number of rows in a database table. Following is the example:

```
sqlite> SELECT count(*) FROM COMPANY;
```

Above SQLite SQL statement will produce the following result:

```
count(*)
----------
7
```

SQLite MAX Function

The SQLite MAX aggregate function allows us to select the highest (maximum) value for a certain column. Following is the example:

```
sqlite> SELECT max(salary) FROM COMPANY;
```

Above SQLite SQL statement will produce the following result:

```
max(salary)
-----------
85000.0
```

SQLite MIN Function

The SQLite MIN aggregate function allows us to select the lowest (minimum) value for a certain column. Following is the example:

```
sqlite> SELECT min(salary) FROM COMPANY;
```

Above SQLite SQL statement will produce the following result:

```
min(salary)
-----------
10000.0
```

SQLite AVG Function

The SQLite AVG aggregate function selects the average value for certain table column. Following is the example:

```
sqlite> SELECT avg(salary) FROM COMPANY;
```

Above SQLite SQL statement will produce the following result:

```
avg(salary)
----------------
37142.8571428572
```

SQLite SUM Function

The SQLite SUM aggregate function allows selecting the total for a numeric column. Following is the example:

```
sqlite> SELECT sum(salary) FROM COMPANY;
```

Above SQLite SQL statement will produce the following result:

```
sum(salary)
-----------
260000.0
```

SQLite RANDOM Function

The SQLite RANDOM function returns a pseudo-random integer between -9223372036854775808 and +9223372036854775807. Following is the example:

```
sqlite> SELECT random() AS Random;
```

Above SQLite SQL statement will produce the following result:

```
Random
-------------------
5876796417670984050
```

SQLite ABS Function

The SQLite ABS function returns the absolute value of the numeric argument. Following is the example:

```
sqlite> SELECT abs(5), abs(-15), abs(NULL), abs(0), abs("ABC");
```

Above SQLite SQL statement will produce following result:

```
abs(5)      abs(-15)    abs(NULL)   abs(0)      abs("ABC")
----------  ----------  ----------  ----------  ----------
5           15                      0           0.0
```

SQLite UPPER Function

The SQLite UPPER function converts a string into upper-case letters. Following is the example:

```
sqlite> SELECT upper(name) FROM COMPANY;
```

Above SQLite SQL statement will produce the following result:

```
upper(name)
```

```
-----------
PAUL
ALLEN
TEDDY
MARK
DAVID
KIM
JAMES
```

SQLite LOWER Function

The SQLite LOWER function converts a string into lower-case letters. Following is the example:

```
sqlite> SELECT lower(name) FROM COMPANY;
```

Above SQLite SQL statement will produce the following result:

```
lower(name)
-----------
paul
allen
teddy
mark
david
kim
james
```

SQLite LENGTH Function

The SQLite LENGTH function returns the length of a string. Following is the example:

```
sqlite> SELECT name, length(name) FROM COMPANY;
```

Above SQLite SQL statement will produce the following result:

```
NAME        length(name)
----------  ------------
Paul        4
Allen       5
Teddy       5
Mark        4
David       5
Kim         3
James       5
```

SQLite sqlite_version Function

The SQLite sqlite_version function returns the version of the SQLite library. Following is the example:

```
sqlite> SELECT sqlite_version() AS 'SQLite Version';
```

Above SQLite SQL statement will produce the following result:

```
SQLite Version
--------------
3.6.20
```

CHAPTER

46

SQLite C/C++ Tutorial

Installation

Before we start using SQLite in our C/C++ programs, we need to make sure that we have SQLite library set up on the machine. You can check SQLite Installation chapter to understand installation process.

C/C++ Interface APIs

Following are important SQLite routines, which can suffice your requirement to work with SQLite database from your C/C++ program. If you are looking for a more sophisticated application, then you can look into SQLite official documentation.

S.N.	API & Description
1	**sqlite3_open(const char *filename, sqlite3 **ppDb)** This routine opens a connection to an SQLite database file and returns a database connection object to be used by other SQLite routines. If the filename argument is NULL or ':memory:', sqlite3_open() will create an in-memory database in RAM that lasts only for the duration of the session. If filename is not NULL, sqlite3_open() attempts to open the database file by using its value. If no file by that name exists, sqlite3_open() will open a new database file by that name.
2	**sqlite3_exec(sqlite3*, const char *sql, sqlite_callback, void *data, char **errmsg)** This routine provides a quick, easy way to execute SQL commands provided by sql argument which can consist of more than one SQL command. Here, first argument sqlite3 is open database object, sqlite_callback is a call back for whichdata is the 1st argument and errmsg will be return to capture any error raised by the routine. The sqlite3_exec() routine parses and executes every command given in the **sql** argument until it reaches the end of the string or encounters an error.
3	**sqlite3_close(sqlite3*)** This routine closes a database connection previously opened by a call to sqlite3_open(). All prepared

	statements associated with the connection should be finalized prior to closing the connection. If any queries remain that have not been finalized, sqlite3_close() will return SQLITE_BUSY with the error message Unable to close due to unfinalized statements.

Connecting To Database

Following C code segment shows how to connect to an existing database. If database does not exist, then it will be created and finally a database object will be returned.

```
#include <stdio.h>
#include <sqlite3.h>

int main(int argc, char* argv[])
{
   sqlite3 *db;
   char *zErrMsg = 0;
   int rc;

   rc = sqlite3_open("test.db", &db);

   if( rc ){
      fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(db));
      exit(0);
   }else{
      fprintf(stderr, "Opened database successfully\n");
   }
   sqlite3_close(db);
}
```

Now, let's compile and run above program to create our database **test.db** in the current directory. You can change your path as per your requirement.

```
$gcc test.c -l sqlite3
$./a.out
Opened database successfully
```

If you are going to use C++ source code, then you can compile your code as follows:

```
$g++ test.c -l sqlite3
```

Here, we are linking our program with sqlite3 library to provide required functions to C program. This will create a database file test.db in your directory and you will have the result something as follows:

```
-rwxr-xr-x. 1 root root 7383 May  8 02:06 a.out
-rw-r--r--. 1 root root  323 May  8 02:05 test.c
-rw-r--r--. 1 root root    0 May  8 02:06 test.db
```

Create a Table

Following C code segment will be used to create a table in previously created database:

```
#include <stdio.h>
#include <stdlib.h>
#include <sqlite3.h>

static int callback(void *NotUsed, int argc, char **argv, char **azColName){
```

```
   int i;
   for(i=0; i<argc; i++){
      printf("%s = %s\n", azColName[i], argv[i] ? argv[i] : "NULL");
   }
   printf("\n");
   return 0;
}

int main(int argc, char* argv[])
{
   sqlite3 *db;
   char *zErrMsg = 0;
   int  rc;
   char *sql;

   /* Open database */
   rc = sqlite3_open("test.db", &db);
   if( rc ){
      fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(db));
      exit(0);
   }else{
      fprintf(stdout, "Opened database successfully\n");
   }

   /* Create SQL statement */
   sql = "CREATE TABLE COMPANY("  \
         "ID INT PRIMARY KEY     NOT NULL," \
         "NAME           TEXT    NOT NULL," \
         "AGE            INT     NOT NULL," \
         "ADDRESS        CHAR(50)," \
         "SALARY         REAL );";

   /* Execute SQL statement */
   rc = sqlite3_exec(db, sql, callback, 0, &zErrMsg);
   if( rc != SQLITE_OK ){
   fprintf(stderr, "SQL error: %s\n", zErrMsg);
      sqlite3_free(zErrMsg);
   }else{
      fprintf(stdout, "Table created successfully\n");
   }
   sqlite3_close(db);
   return 0;
}
```

When above program is compiled and executed, it will create COMPANY table in your test.db and final listing of the file will be as follows:

```
-rwxr-xr-x. 1 root root 9567 May  8 02:31 a.out
-rw-r--r--. 1 root root 1207 May  8 02:31 test.c
-rw-r--r--. 1 root root 3072 May  8 02:31 test.db
```

INSERT Operation

Following C code segment shows how we can create records in our COMPANY table created in above example:

```
#include <stdio.h>
#include <stdlib.h>
#include <sqlite3.h>
```

```
static int callback(void *NotUsed, int argc, char **argv, char **azColName){
   int i;
   for(i=0; i<argc; i++){
      printf("%s = %s\n", azColName[i], argv[i] ? argv[i] : "NULL");
   }
   printf("\n");
   return 0;
}

int main(int argc, char* argv[])
{
   sqlite3 *db;
   char *zErrMsg = 0;
   int rc;
   char *sql;

   /* Open database */
   rc = sqlite3_open("test.db", &db);
   if( rc ){
      fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(db));
      exit(0);
   }else{
      fprintf(stderr, "Opened database successfully\n");
   }

   /* Create SQL statement */
   sql = "INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) "  \
         "VALUES (1, 'Paul', 32, 'California', 20000.00 ); " \
         "INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) "  \
         "VALUES (2, 'Allen', 25, 'Texas', 15000.00 ); "     \
         "INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)" \
         "VALUES (3, 'Teddy', 23, 'Norway', 20000.00 );" \
         "INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)" \
         "VALUES (4, 'Mark', 25, 'Rich-Mond ', 65000.00 );";

   /* Execute SQL statement */
   rc = sqlite3_exec(db, sql, callback, 0, &zErrMsg);
   if( rc != SQLITE_OK ){
      fprintf(stderr, "SQL error: %s\n", zErrMsg);
      sqlite3_free(zErrMsg);
   }else{
      fprintf(stdout, "Records created successfully\n");
   }
   sqlite3_close(db);
   return 0;
}
```

When above program is compiled and executed, it will create given records in COMPANY table and will display following two line:

```
Opened database successfully
Records created successfully
```

SELECT Operation

Before we proceed with actual example to fetch records, let me give a little detail about the callback function, which we are using in our examples. This callback provides a way to obtain results from SELECT statements. It has the following declaration:

```
typedef int (*sqlite3_callback)(
```

```
void*,    /* Data provided in the 4th argument of sqlite3_exec() */
int,      /* The number of columns in row */
char**,   /* An array of strings representing fields in the row */
char**    /* An array of strings representing column names */
);
```

If above callback is provided in sqlite_exec() routine as the third argument, SQLite will call the this callback function for each record processed in each SELECT statement executed within the SQL argument.

Following C code segment shows how we can fetch and display records from our COMPANY table created in above example:

```
#include <stdio.h>
#include <stdlib.h>
#include <sqlite3.h>

static int callback(void *data, int argc, char **argv, char **azColName){
   int i;
   fprintf(stderr, "%s: ", (const char*)data);
   for(i=0; i<argc; i++){
      printf("%s = %s\n", azColName[i], argv[i] ? argv[i] : "NULL");
   }
   printf("\n");
   return 0;
}

int main(int argc, char* argv[])
{
   sqlite3 *db;
   char *zErrMsg = 0;
   int rc;
   char *sql;
   const char* data = "Callback function called";

   /* Open database */
   rc = sqlite3_open("test.db", &db);
   if( rc ){
      fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(db));
      exit(0);
   }else{
      fprintf(stderr, "Opened database successfully\n");
   }

   /* Create SQL statement */
   sql = "SELECT * from COMPANY";

   /* Execute SQL statement */
   rc = sqlite3_exec(db, sql, callback, (void*)data, &zErrMsg);
   if( rc != SQLITE_OK ){
      fprintf(stderr, "SQL error: %s\n", zErrMsg);
      sqlite3_free(zErrMsg);
   }else{
      fprintf(stdout, "Operation done successfully\n");
   }
   sqlite3_close(db);
   return 0;
}
```

When above program is compiled and executed, it will produce the following result:

```
Opened database successfully
Callback function called: ID = 1
NAME = Paul
AGE = 32
ADDRESS = California
SALARY = 20000.0

Callback function called: ID = 2
NAME = Allen
AGE = 25
ADDRESS = Texas
SALARY = 15000.0

Callback function called: ID = 3
NAME = Teddy
AGE = 23
ADDRESS = Norway
SALARY = 20000.0

Callback function called: ID = 4
NAME = Mark
AGE = 25
ADDRESS = Rich-Mond
SALARY = 65000.0

Operation done successfully
```

UPDATE Operation

Following C code segment shows how we can use UPDATE statement to update any record and then fetch and display updated records from our COMPANY table:

```
#include <stdio.h>
#include <stdlib.h>
#include <sqlite3.h>

static int callback(void *data, int argc, char **argv, char **azColName){
   int i;
   fprintf(stderr, "%s: ", (const char*)data);
   for(i=0; i<argc; i++){
      printf("%s = %s\n", azColName[i], argv[i] ? argv[i] : "NULL");
   }
   printf("\n");
   return 0;
}

int main(int argc, char* argv[])
{
   sqlite3 *db;
   char *zErrMsg = 0;
   int rc;
   char *sql;
   const char* data = "Callback function called";

   /* Open database */
   rc = sqlite3_open("test.db", &db);
   if( rc ){
      fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(db));
      exit(0);
   }else{
```

```
      fprintf(stderr, "Opened database successfully\n");
   }

   /* Create merged SQL statement */
   sql = "UPDATE COMPANY set SALARY = 25000.00 where ID=1; " \
         "SELECT * from COMPANY";

   /* Execute SQL statement */
   rc = sqlite3_exec(db, sql, callback, (void*)data, &zErrMsg);
   if( rc != SQLITE_OK ){
      fprintf(stderr, "SQL error: %s\n", zErrMsg);
      sqlite3_free(zErrMsg);
   }else{
      fprintf(stdout, "Operation done successfully\n");
   }
   sqlite3_close(db);
   return 0;
}
```

When above program is compiled and executed, it will produce the following result:

```
Opened database successfully
Callback function called: ID = 1
NAME = Paul
AGE = 32
ADDRESS = California
SALARY = 25000.0

Callback function called: ID = 2
NAME = Allen
AGE = 25
ADDRESS = Texas
SALARY = 15000.0

Callback function called: ID = 3
NAME = Teddy
AGE = 23
ADDRESS = Norway
SALARY = 20000.0

Callback function called: ID = 4
NAME = Mark
AGE = 25
ADDRESS = Rich-Mond
SALARY = 65000.0

Operation done successfully
```

DELETE Operation

Following C code segment shows how we can use DELETE statement to delete any record and then fetch and display remaining records from our COMPANY table:

```
#include <stdio.h>
#include <stdlib.h>
#include <sqlite3.h>

static int callback(void *data, int argc, char **argv, char **azColName){
   int i;
   fprintf(stderr, "%s: ", (const char*)data);
```

```
   for(i=0; i<argc; i++){
      printf("%s = %s\n", azColName[i], argv[i] ? argv[i] : "NULL");
   }
   printf("\n");
   return 0;
}

int main(int argc, char* argv[])
{
   sqlite3 *db;
   char *zErrMsg = 0;
   int rc;
   char *sql;
   const char* data = "Callback function called";

   /* Open database */
   rc = sqlite3_open("test.db", &db);
   if( rc ){
      fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(db));
      exit(0);
   }else{
      fprintf(stderr, "Opened database successfully\n");
   }

   /* Create merged SQL statement */
   sql = "DELETE from COMPANY where ID=2; " \
         "SELECT * from COMPANY";

   /* Execute SQL statement */
   rc = sqlite3_exec(db, sql, callback, (void*)data, &zErrMsg);
   if( rc != SQLITE_OK ){
      fprintf(stderr, "SQL error: %s\n", zErrMsg);
      sqlite3_free(zErrMsg);
   }else{
      fprintf(stdout, "Operation done successfully\n");
   }
   sqlite3_close(db);
   return 0;
}
```

When above program is compiled and executed, it will produce the following result:

```
Opened database successfully
Callback function called: ID = 1
NAME = Paul
AGE = 32
ADDRESS = California
SALARY = 20000.0

Callback function called: ID = 3
NAME = Teddy
AGE = 23
ADDRESS = Norway
SALARY = 20000.0

Callback function called: ID = 4
NAME = Mark
AGE = 25
ADDRESS = Rich-Mond
SALARY = 65000.0
```

```
Operation done successfully
```

CHAPTER 47

SQLite Java Tutorial

Installation

Before we start using SQLite in our Java programs, we need to make sure that we have SQLite JDBC Driver and Java set up on the machine. You can check Java tutorial for Java installation on your machine. Now, let us check how to set up SQLite JDBC driver.

- Download latest version of sqlite-jdbc-(VERSION).jar from sqlite-jdbc repository.
- Add downloaded jar file sqlite-jdbc-(VERSION).jar in your class path, or you can use it along with -classpath option as explained below in examples.

Following section assumes you have little knowledge about Java JDBC concepts. If you don't, then it is suggested to spent half an hour with JDBC Tutorial to become comfortable with concepts explained below.

Connecting To Database

Following Java programs shows how to connect to an existing database. If database does not exist, then it will be created and finally a database object will be returned.

```
import java.sql.*;

public class SQLiteJDBC
{
  public static void main( String args[] )
  {
    Connection c = null;
    try {
      Class.forName("org.sqlite.JDBC");
      c = DriverManager.getConnection("jdbc:sqlite:test.db");
    } catch ( Exception e ) {
      System.err.println( e.getClass().getName() + ": " + e.getMessage() );
      System.exit(0);
    }
    System.out.println("Opened database successfully");
  }
}
```

Now, let's compile and run above program to create our database **test.db** in the current directory. You can change your path as per your requirement. We are assuming current version of JDBC driver sqlite-jdbc-3.7.2.jar is available in the current path

```
$javac SQLiteJDBC.java
$java -classpath ".:sqlite-jdbc-3.7.2.jar" SQLiteJDBC
Open database successfully
```

If you are going to use Windows machine, then you can compile and run your code as follows:

```
$javac SQLiteJDBC.java
$java -classpath ".;sqlite-jdbc-3.7.2.jar" SQLiteJDBC
Opened database successfully
```

Create a Table

Following Java program will be used to create a table in previously created database:

```
import java.sql.*;

public class SQLiteJDBC
{
  public static void main( String args[] )
  {
    Connection c = null;
    Statement stmt = null;
    try {
      Class.forName("org.sqlite.JDBC");
      c = DriverManager.getConnection("jdbc:sqlite:test.db");
      System.out.println("Opened database successfully");

      stmt = c.createStatement();
      String sql = "CREATE TABLE COMPANY " +
                   "(ID INT PRIMARY KEY     NOT NULL," +
                   " NAME           TEXT    NOT NULL, " +
                   " AGE            INT     NOT NULL, " +
                   " ADDRESS        CHAR(50), " +
                   " SALARY         REAL)";
      stmt.executeUpdate(sql);
      stmt.close();
      c.close();
    } catch ( Exception e ) {
      System.err.println( e.getClass().getName() + ": " + e.getMessage() );
      System.exit(0);
    }
    System.out.println("Table created successfully");
  }
}
```

When above program is compiled and executed, it will create COMPANY table in your **test.db** and final listing of the file will be as follows:

```
-rw-r--r--. 1 root root 3201128 Jan 22 19:04 sqlite-jdbc-3.7.2.jar
-rw-r--r--. 1 root root    1506 May  8 05:43 SQLiteJDBC.class
-rw-r--r--. 1 root root     832 May  8 05:42 SQLiteJDBC.java
-rw-r--r--. 1 root root    3072 May  8 05:43 test.db
```

INSERT Operation

Following Java program shows how we can create records in our COMPANY table created in above example:

```
import java.sql.*;
```

```
public class SQLiteJDBC
{
  public static void main( String args[] )
  {
    Connection c = null;
    Statement stmt = null;
    try {
      Class.forName("org.sqlite.JDBC");
      c = DriverManager.getConnection("jdbc:sqlite:test.db");
      c.setAutoCommit(false);
      System.out.println("Opened database successfully");

      stmt = c.createStatement();
      String sql = "INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) " +
                   "VALUES (1, 'Paul', 32, 'California', 20000.00 );";
      stmt.executeUpdate(sql);

      sql = "INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) " +
            "VALUES (2, 'Allen', 25, 'Texas', 15000.00 );";
      stmt.executeUpdate(sql);

      sql = "INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) " +
            "VALUES (3, 'Teddy', 23, 'Norway', 20000.00 );";
      stmt.executeUpdate(sql);

      sql = "INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) " +
            "VALUES (4, 'Mark', 25, 'Rich-Mond ', 65000.00 );";
      stmt.executeUpdate(sql);

      stmt.close();
      c.commit();
      c.close();
    } catch ( Exception e ) {
      System.err.println( e.getClass().getName() + ": " + e.getMessage() );
      System.exit(0);
    }
    System.out.println("Records created successfully");
  }
}
```

When above program is compiled and executed, it will create given records in COMPANY table and will display following two line:

```
Opened database successfully
Records created successfully
```

SELECT Operation

Following Java program shows how we can fetch and display records from our COMPANY table created in above example:

```
import java.sql.*;

public class SQLiteJDBC
{
  public static void main( String args[] )
  {
    Connection c = null;
    Statement stmt = null;
```

```
    try {
      Class.forName("org.sqlite.JDBC");
      c = DriverManager.getConnection("jdbc:sqlite:test.db");
      c.setAutoCommit(false);
      System.out.println("Opened database successfully");

      stmt = c.createStatement();
      ResultSet rs = stmt.executeQuery( "SELECT * FROM COMPANY;" );
      while ( rs.next() ) {
         int id = rs.getInt("id");
         String  name = rs.getString("name");
         int age  = rs.getInt("age");
         String  address = rs.getString("address");
         float salary = rs.getFloat("salary");
         System.out.println( "ID = " + id );
         System.out.println( "NAME = " + name );
         System.out.println( "AGE = " + age );
         System.out.println( "ADDRESS = " + address );
         System.out.println( "SALARY = " + salary );
         System.out.println();
      }
      rs.close();
      stmt.close();
      c.close();
    } catch ( Exception e ) {
      System.err.println( e.getClass().getName() + ": " + e.getMessage() );
      System.exit(0);
    }
    System.out.println("Operation done successfully");
  }
}
```

When above program is compiled and executed, it will produce the following result:

```
Opened database successfully
ID = 1
NAME = Paul
AGE = 32
ADDRESS = California
SALARY = 20000.0

ID = 2
NAME = Allen
AGE = 25
ADDRESS = Texas
SALARY = 15000.0

ID = 3
NAME = Teddy
AGE = 23
ADDRESS = Norway
SALARY = 20000.0

ID = 4
NAME = Mark
AGE = 25
ADDRESS = Rich-Mond
SALARY = 65000.0

Operation done successfully
```

UPDATE Operation

Following Java code shows how we can use UPDATE statement to update any record and then fetch and display updated records from our COMPANY table:

```
import java.sql.*;

public class SQLiteJDBC
{
  public static void main( String args[] )
  {
    Connection c = null;
    Statement stmt = null;
    try {
      Class.forName("org.sqlite.JDBC");
      c = DriverManager.getConnection("jdbc:sqlite:test.db");
      c.setAutoCommit(false);
      System.out.println("Opened database successfully");

      stmt = c.createStatement();
      String sql = "UPDATE COMPANY set SALARY = 25000.00 where ID=1;";
      stmt.executeUpdate(sql);
      c.commit();

      ResultSet rs = stmt.executeQuery( "SELECT * FROM COMPANY;" );
      while ( rs.next() ) {
         int id = rs.getInt("id");
         String  name = rs.getString("name");
         int age  = rs.getInt("age");
         String  address = rs.getString("address");
         float salary = rs.getFloat("salary");
         System.out.println( "ID = " + id );
         System.out.println( "NAME = " + name );
         System.out.println( "AGE = " + age );
         System.out.println( "ADDRESS = " + address );
         System.out.println( "SALARY = " + salary );
         System.out.println();
      }
      rs.close();
      stmt.close();
      c.close();
    } catch ( Exception e ) {
      System.err.println( e.getClass().getName() + ": " + e.getMessage() );
      System.exit(0);
    }
    System.out.println("Operation done successfully");
  }
}
```

When above program is compiled and executed, it will produce the following result:

```
Opened database successfully
ID = 1
NAME = Paul
AGE = 32
ADDRESS = California
SALARY = 25000.0

ID = 2
NAME = Allen
```

```
AGE = 25
ADDRESS = Texas
SALARY = 15000.0

ID = 3
NAME = Teddy
AGE = 23
ADDRESS = Norway
SALARY = 20000.0

ID = 4
NAME = Mark
AGE = 25
ADDRESS = Rich-Mond
SALARY = 65000.0

Operation done successfully
```

DELETE Operation

Following Java code shows how we can use DELETE statement to delete any record and then fetch and display remaining records from our COMPANY table:

```
import java.sql.*;

public class SQLiteJDBC
{
  public static void main( String args[] )
  {
    Connection c = null;
    Statement stmt = null;
    try {
      Class.forName("org.sqlite.JDBC");
      c = DriverManager.getConnection("jdbc:sqlite:test.db");
      c.setAutoCommit(false);
      System.out.println("Opened database successfully");

      stmt = c.createStatement();
      String sql = "DELETE from COMPANY where ID=2;";
      stmt.executeUpdate(sql);
      c.commit();

      ResultSet rs = stmt.executeQuery( "SELECT * FROM COMPANY;" );
      while ( rs.next() ) {
         int id = rs.getInt("id");
         String  name = rs.getString("name");
         int age  = rs.getInt("age");
         String  address = rs.getString("address");
         float salary = rs.getFloat("salary");
         System.out.println( "ID = " + id );
         System.out.println( "NAME = " + name );
         System.out.println( "AGE = " + age );
         System.out.println( "ADDRESS = " + address );
         System.out.println( "SALARY = " + salary );
         System.out.println();
      }
      rs.close();
      stmt.close();
      c.close();
    } catch ( Exception e ) {
```

```
      System.err.println( e.getClass().getName() + ": " + e.getMessage() );
      System.exit(0);
    }
    System.out.println("Operation done successfully");
  }
}
```

When above program is compiled and executed, it will produce the following result:

```
Opened database successfully
ID = 1
NAME = Paul
AGE = 32
ADDRESS = California
SALARY = 25000.0

ID = 3
NAME = Teddy
AGE = 23
ADDRESS = Norway
SALARY = 20000.0

ID = 4
NAME = Mark
AGE = 25
ADDRESS = Rich-Mond
SALARY = 65000.0

Operation done successfully
```

CHAPTER

48

SQLite PHP Tutorial

Installation

The SQLite3 extension is enabled by default as of PHP 5.3.0. It's possible to disable it by using **--without-sqlite3** at compile time.

Windows users must enable php_sqlite3.dll in order to use this extension. This DLL is included with Windows distributions of PHP as of PHP 5.3.0.

For detailed installation instructions, kindly check our PHP tutorial and its official website.

PHP Interface APIs

Following are important PHP routines, which can suffice your requirement to work with SQLite database from your PHP program. If you are looking for a more sophisticated application, then you can look into PHP official documentation.

S.N.	API & Description
1	**public void SQLite3::open (filename, flags, encryption_key)** Opens an SQLite 3 Database. If the build includes encryption, then it will attempt to use the key. If the filename is given as **':memory:'**, SQLite3::open() will create an in-memory database in RAM that lasts only for the duration of the session. If filename is actual device file name, SQLite3::open() attempts to open the database file by using its value. If no file by that name exists then a new database file by that name gets created. Optional flags used to determine how to open the SQLite database. By default, open uses SQLITE3_OPEN_READWRITE | SQLITE3_OPEN_CREATE.
2	**public bool SQLite3::exec (string $query)** This routine provides a quick, easy way to execute SQL commands provided by sql argument which can consist of more than one SQL command. This routine is used to execute a result-less query against a given database.

3	**public SQLite3Result SQLite3::query (string $query)** This routine executes an SQL query, returning an **SQLite3Result** object if the query returns results.
4	**public int SQLite3::lastErrorCode (void)** This routine returns the numeric result code of the most recent failed SQLite request
5	**public string SQLite3::lastErrorMsg (void)** This routine returns english text describing the most recent failed SQLite request.
6	**public int SQLite3::changes (void)** This routine returns the number of database rows that were updated or inserted or deleted by the most recent SQL statement
7	**public bool SQLite3::close (void)** This routine closes a database connection previously opened by a call to SQLite3::open().
8	**public string SQLite3::escapeString (string $value)** This routine returns a string that has been properly escaped for safe inclusion in an SQL statement.

Connecting To Database

Following PHP code shows how to connect to an existing database. If database does not exist, then it will be created and finally a database object will be returned.

```
<?php
   class MyDB extends SQLite3
   {
      function __construct()
      {
         $this->open('test.db');
      }
   }
   $db = new MyDB();
   if(!$db){
      echo $db->lastErrorMsg();
   } else {
      echo "Opened database successfully\n";
   }
?>
```

Now, let's run above program to create our database **test.db** in the current directory. You can change your path as per your requirement. If database is successfully created, then it will give the following message:

```
Open database successfully
```

Create a Table

Following PHP program will be used to create a table in previously created database:

```
<?php
   class MyDB extends SQLite3
   {
      function __construct()
      {
         $this->open('test.db');
      }
   }
   $db = new MyDB();
   if(!$db){
      echo $db->lastErrorMsg();
   } else {
      echo "Opened database successfully\n";
   }

   $sql =<<<EOF
      CREATE TABLE COMPANY
      (ID INT PRIMARY KEY     NOT NULL,
      NAME           TEXT    NOT NULL,
      AGE            INT     NOT NULL,
      ADDRESS        CHAR(50),
      SALARY         REAL);
EOF;

   $ret = $db->exec($sql);
   if(!$ret){
      echo $db->lastErrorMsg();
   } else {
      echo "Table created successfully\n";
   }
   $db->close();
?>
```

When above program is executed, it will create COMPANY table in your **test.db** and it will display the following messages:

```
Opened database successfully
Table created successfully
```

INSERT Operation

Following PHP program shows how we can create records in our COMPANY table created in above example:

```
<?php
   class MyDB extends SQLite3
   {
      function __construct()
      {
         $this->open('test.db');
      }
   }
   $db = new MyDB();
   if(!$db){
      echo $db->lastErrorMsg();
   } else {
      echo "Opened database successfully\n";
   }

   $sql =<<<EOF
```

```
        INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
        VALUES (1, 'Paul', 32, 'California', 20000.00 );

        INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
        VALUES (2, 'Allen', 25, 'Texas', 15000.00 );

        INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
        VALUES (3, 'Teddy', 23, 'Norway', 20000.00 );

        INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
        VALUES (4, 'Mark', 25, 'Rich-Mond ', 65000.00 );
EOF;

   $ret = $db->exec($sql);
   if(!$ret){
      echo $db->lastErrorMsg();
   } else {
      echo "Records created successfully\n";
   }
   $db->close();
?>
```

When above program is executed, it will create given records in COMPANY table and will display the following two lines:

```
Opened database successfully
Records created successfully
```

SELECT Operation

Following PHP program shows how we can fetch and display records from our COMPANY table created in above example:

```
<?php
   class MyDB extends SQLite3
   {
      function __construct()
      {
         $this->open('test.db');
      }
   }
   $db = new MyDB();
   if(!$db){
      echo $db->lastErrorMsg();
   } else {
      echo "Opened database successfully\n";
   }

   $sql =<<<EOF
      SELECT * from COMPANY;
EOF;

   $ret = $db->query($sql);
   while($row = $ret->fetchArray(SQLITE3_ASSOC) ){
      echo "ID = ". $row['ID'] . "\n";
      echo "NAME = ". $row['NAME'] ."\n";
      echo "ADDRESS = ". $row['ADDRESS'] ."\n";
      echo "SALARY =  ".$row['SALARY'] ."\n\n";
   }
   echo "Operation done successfully\n";
```

```
   $db->close();
?>
```

When above program is executed, it will produce the following result:

```
Opened database successfully
ID = 1
NAME = Paul
ADDRESS = California
SALARY =  20000

ID = 2
NAME = Allen
ADDRESS = Texas
SALARY =  15000

ID = 3
NAME = Teddy
ADDRESS = Norway
SALARY =  20000

ID = 4
NAME = Mark
ADDRESS = Rich-Mond
SALARY =  65000

Operation done successfully
```

UPDATE Operation

Following PHP code shows how we can use UPDATE statement to update any record and then fetch and display updated records from our COMPANY table:

```
<?php
   class MyDB extends SQLite3
   {
      function __construct()
      {
         $this->open('test.db');
      }
   }
   $db = new MyDB();
   if(!$db){
      echo $db->lastErrorMsg();
   } else {
      echo "Opened database successfully\n";
   }
   $sql =<<<EOF
      UPDATE COMPANY set SALARY = 25000.00 where ID=1;
EOF;
   $ret = $db->exec($sql);
   if(!$ret){
      echo $db->lastErrorMsg();
   } else {
      echo $db->changes(), " Record updated successfully\n";
   }

   $sql =<<<EOF
      SELECT * from COMPANY;
EOF;
```

```
   $ret = $db->query($sql);
   while($row = $ret->fetchArray(SQLITE3_ASSOC) ){
      echo "ID = ". $row['ID'] . "\n";
      echo "NAME = ". $row['NAME'] ."\n";
      echo "ADDRESS = ". $row['ADDRESS'] ."\n";
      echo "SALARY =  ".$row['SALARY'] ."\n\n";
   }
   echo "Operation done successfully\n";
   $db->close();
?>
```

When above program is executed, it will produce the following result:

```
Opened database successfully
1 Record updated successfully
ID = 1
NAME = Paul
ADDRESS = California
SALARY =  25000

ID = 2
NAME = Allen
ADDRESS = Texas
SALARY =  15000

ID = 3
NAME = Teddy
ADDRESS = Norway
SALARY =  20000

ID = 4
NAME = Mark
ADDRESS = Rich-Mond
SALARY =  65000

Operation done successfully
```

DELETE Operation

Following PHP code shows how we can use DELETE statement to delete any record and then fetch and display remaining records from our COMPANY table:

```
<?php
   class MyDB extends SQLite3
   {
      function __construct()
      {
         $this->open('test.db');
      }
   }
   $db = new MyDB();
   if(!$db){
      echo $db->lastErrorMsg();
   } else {
      echo "Opened database successfully\n";
   }
   $sql =<<<EOF
      DELETE from COMPANY where ID=2;
EOF;
   $ret = $db->exec($sql);
```

```
   if(!$ret){
     echo $db->lastErrorMsg();
   } else {
      echo $db->changes(), " Record deleted successfully\n";
   }

   $sql =<<<EOF
      SELECT * from COMPANY;
EOF;
   $ret = $db->query($sql);
   while($row = $ret->fetchArray(SQLITE3_ASSOC) ){
      echo "ID = ". $row['ID'] . "\n";
      echo "NAME = ". $row['NAME'] ."\n";
      echo "ADDRESS = ". $row['ADDRESS'] ."\n";
      echo "SALARY =  ".$row['SALARY'] ."\n\n";
   }
   echo "Operation done successfully\n";
   $db->close();
?>
```

When above program is executed, it will produce the following result:

```
Opened database successfully
1 Record deleted successfully
ID = 1
NAME = Paul
ADDRESS = California
SALARY =  25000

ID = 3
NAME = Teddy
ADDRESS = Norway
SALARY =  20000

ID = 4
NAME = Mark
ADDRESS = Rich-Mond
SALARY =  65000

Operation done successfully
```

CHAPTER

49

SQLite Perl Tutorial

Installation

The SQLite3 can be integrated with Perl using Perl DBI module, which is a database access module for the Perl programming language. It defines a set of methods, variables and conventions that provide a standard database interface.

Here are simple steps to install DBI module on your Linux/UNIX machine:

```
$ wget http://search.cpan.org/CPAN/authors/id/T/TI/TIMB/DBI-1.625.tar.gz
$ tar xvfz DBI-1.625.tar.gz
$ cd DBI-1.625
$ perl Makefile.PL
$ make
$ make install
```

If you need to install SQLite driver for DBI, then it can be installed as follows:

```
$ wget http://search.cpan.org/CPAN/authors/id/M/MS/MSERGEANT/DBD-SQLite-1.11.tar.gz
$ tar xvfz DBD-SQLite-1.11.tar.gz
$ cd DBD-SQLite-1.11
$ perl Makefile.PL
$ make
$ make install
```

DBI Interface APIs

Following are important DBI routines, which can suffice your requirement to work with SQLite database from your Perl program. If you are looking for a more sophisticated application, then you can look into Perl DBI official documentation.

S.N.	API & Description
1	**DBI->connect($data_source, "", "", \%attr)** Establishes a database connection, or session, to the requested $data_source. Returns a database handle object if the connection succeeds. Datasource has the form like : **DBI:SQLite:dbname='test.db'** SQLite is SQLite driver name and

	test.db is the name of SQLite database file. If the filename is given as **':memory:'**, it will create an in-memory database in RAM that lasts only for the duration of the session. If filename is actual device file name, then it attempts to open the database file by using its value. If no file by that name exists then a new database file by that name gets created. You keep second and third paramter as blank strings and last parameter is to pass various attributes as shown below in the example.
2	**$dbh->do($sql)** This routine prepares and executes a single SQL statement. Returns the number of rows affected or undef on error. A return value of -1 means the number of rows is not known, not applicable, or not available. Here $dbh is a handle returned by DBI->connect() call.
3	**$dbh->prepare($sql)** This routine prepares a statement for later execution by the database engine and returns a reference to a statement handle object.
4	**$sth->execute()** This routine performs whatever processing is necessary to execute the prepared statement. An undef is returned if an error occurs. A successful execute always returns true regardless of the number of rows affected. Here, $sth is a statement handle returned by $dbh->prepare($sql) call.
5	**$sth->fetchrow_array()** This routine fetches the next row of data and returns it as a list containing the field values. Null fields are returned as undef values in the list.
6	**$DBI::err** This is equivalent to $h->err, where $h is any of the handle types like $dbh, $sth, or $drh. This returns native database engine error code from the last driver method called.
7	**$DBI::errstr** This is equivalent to $h->errstr, where $h is any of the handle types like $dbh, $sth, or $drh. This returns the native database engine error message from the last DBI method called.
8	**$dbh->disconnect()** This routine closes a database connection previously opened by a call to DBI->connect().

Connecting To Database

Following Perl code shows how to connect to an existing database. If database does not exist, then it will be created and finally a database object will be returned.

```
#!/usr/bin/perl
```

```
use DBI;
use strict;

my $driver   = "SQLite";
my $database = "test.db";
my $dsn = "DBI:$driver:dbname=$database";
my $userid = "";
my $password = "";
my $dbh = DBI->connect($dsn, $userid, $password, { RaiseError => 1 })
                      or die $DBI::errstr;

print "Opened database successfully\n";
```

Now, let's run above program to create our database **test.db** in the current directory. You can change your path as per your requirement. Keep above code in sqlite.pl file and execute it as shown below. If database is successfully created, then it will give the following message:

```
$ chmod +x sqlite.pl
$ ./sqlite.pl
Open database successfully
```

Create a Table

Following Perl program will be used to create a table in previously created database:

```
#!/usr/bin/perl

use DBI;
use strict;

my $driver   = "SQLite";
my $database = "test.db";
my $dsn = "DBI:$driver:dbname=$database";
my $userid = "";
my $password = "";
my $dbh = DBI->connect($dsn, $userid, $password, { RaiseError => 1 })
                      or die $DBI::errstr;
print "Opened database successfully\n";

my $stmt = qq(CREATE TABLE COMPANY
      (ID INT PRIMARY KEY     NOT NULL,
       NAME           TEXT    NOT NULL,
       AGE            INT     NOT NULL,
       ADDRESS        CHAR(50),
       SALARY         REAL););
my $rv = $dbh->do($stmt);
if($rv < 0){
   print $DBI::errstr;
} else {
   print "Table created successfully\n";
}
$dbh->disconnect();
```

When above program is executed, it will create COMPANY table in your **test.db** and it will display the following messages:

```
Opened database successfully
Table created successfully
```

NOTE: in case you see following error in any of the operation:

```
DBD::SQLite::st execute failed: not an error(21) at dbdimp.c line 398
```

In this case you will have open dbdimp.c file available in DBD-SQLite installation and find out **sqlite3_prepare()** function and change its third argument to -1 instead of 0. Finally install DBD::SQLite using **make** and do **make install** to resolve the problem.

INSERT Operation

Following Perl program shows how we can create records in our COMPANY table created in above example:

```
#!/usr/bin/perl

use DBI;
use strict;

my $driver   = "SQLite";
my $database = "test.db";
my $dsn = "DBI:$driver:dbname=$database";
my $userid = "";
my $password = "";
my $dbh = DBI->connect($dsn, $userid, $password, { RaiseError => 1 })
                      or die $DBI::errstr;
print "Opened database successfully\n";

my $stmt = qq(INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
      VALUES (1, 'Paul', 32, 'California', 20000.00 ));
my $rv = $dbh->do($stmt) or die $DBI::errstr;

$stmt = qq(INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
      VALUES (2, 'Allen', 25, 'Texas', 15000.00 ));
$rv = $dbh->do($stmt) or die $DBI::errstr;

$stmt = qq(INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
      VALUES (3, 'Teddy', 23, 'Norway', 20000.00 ));
$rv = $dbh->do($stmt) or die $DBI::errstr;

$stmt = qq(INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
      VALUES (4, 'Mark', 25, 'Rich-Mond ', 65000.00 ););
$rv = $dbh->do($stmt) or die $DBI::errstr;

print "Records created successfully\n";
$dbh->disconnect();
```

When above program is executed, it will create given records in COMPANY table and will display the following two lines:

```
Opened database successfully
Records created successfully
```

SELECT Operation

Following Perl program shows how we can fetch and display records from our COMPANY table created in above example:

```
#!/usr/bin/perl

use DBI;
use strict;
```

```
my $driver   = "SQLite";
my $database = "test.db";
my $dsn = "DBI:$driver:dbname=$database";
my $userid = "";
my $password = "";
my $dbh = DBI->connect($dsn, $userid, $password, { RaiseError => 1 })
                      or die $DBI::errstr;
print "Opened database successfully\n";

my $stmt = qq(SELECT id, name, address, salary  from COMPANY;);
my $sth = $dbh->prepare( $stmt );
my $rv = $sth->execute() or die $DBI::errstr;
if($rv < 0){
   print $DBI::errstr;
}
while(my @row = $sth->fetchrow_array()) {
      print "ID = ". $row[0] . "\n";
      print "NAME = ". $row[1] ."\n";
      print "ADDRESS = ". $row[2] ."\n";
      print "SALARY =  ". $row[3] ."\n\n";
}
print "Operation done successfully\n";
$dbh->disconnect();
```

When above program is executed, it will produce the following result:

```
Opened database successfully
ID = 1
NAME = Paul
ADDRESS = California
SALARY =  20000

ID = 2
NAME = Allen
ADDRESS = Texas
SALARY =  15000

ID = 3
NAME = Teddy
ADDRESS = Norway
SALARY =  20000

ID = 4
NAME = Mark
ADDRESS = Rich-Mond
SALARY =  65000

Operation done successfully
```

UPDATE Operation

Following Perl code shows how we can use UPDATE statement to update any record and then fetch and display updated records from our COMPANY table:

```
#!/usr/bin/perl

use DBI;
use strict;
```

```
my $driver   = "SQLite";
my $database = "test.db";
my $dsn = "DBI:$driver:dbname=$database";
my $userid = "";
my $password = "";
my $dbh = DBI->connect($dsn, $userid, $password, { RaiseError => 1 })
                      or die $DBI::errstr;
print "Opened database successfully\n";

my $stmt = qq(UPDATE COMPANY set SALARY = 25000.00 where ID=1;);
my $rv = $dbh->do($stmt) or die $DBI::errstr;
if( $rv < 0 ){
   print $DBI::errstr;
}else{
   print "Total number of rows updated : $rv\n";
}
$stmt = qq(SELECT id, name, address, salary  from COMPANY;);
my $sth = $dbh->prepare( $stmt );
$rv = $sth->execute() or die $DBI::errstr;
if($rv < 0){
   print $DBI::errstr;
}
while(my @row = $sth->fetchrow_array()) {
      print "ID = ". $row[0] . "\n";
      print "NAME = ". $row[1] ."\n";
      print "ADDRESS = ". $row[2] ."\n";
      print "SALARY =  ". $row[3] ."\n\n";
}
print "Operation done successfully\n";
$dbh->disconnect();
```

When above program is executed, it will produce the following result:

```
Opened database successfully
Total number of rows updated : 1
ID = 1
NAME = Paul
ADDRESS = California
SALARY =  25000

ID = 2
NAME = Allen
ADDRESS = Texas
SALARY =  15000

ID = 3
NAME = Teddy
ADDRESS = Norway
SALARY =  20000

ID = 4
NAME = Mark
ADDRESS = Rich-Mond
SALARY =  65000

Operation done successfully
```

DELETE Operation

Following Perl code shows how we can use DELETE statement to delete any record and then fetch and display remaining records from our COMPANY table:

```
#!/usr/bin/perl

use DBI;
use strict;

my $driver   = "SQLite";
my $database = "test.db";
my $dsn = "DBI:$driver:dbname=$database";
my $userid = "";
my $password = "";
my $dbh = DBI->connect($dsn, $userid, $password, { RaiseError => 1 })
                      or die $DBI::errstr;
print "Opened database successfully\n";

my $stmt = qq(DELETE from COMPANY where ID=2;);
my $rv = $dbh->do($stmt) or die $DBI::errstr;
if( $rv < 0 ){
   print $DBI::errstr;
}else{
   print "Total number of rows deleted : $rv\n";
}
$stmt = qq(SELECT id, name, address, salary  from COMPANY;);
my $sth = $dbh->prepare( $stmt );
$rv = $sth->execute() or die $DBI::errstr;
if($rv < 0){
   print $DBI::errstr;
}
while(my @row = $sth->fetchrow_array()) {
      print "ID = ". $row[0] . "\n";
      print "NAME = ". $row[1] ."\n";
      print "ADDRESS = ". $row[2] ."\n";
      print "SALARY =  ". $row[3] ."\n\n";
}
print "Operation done successfully\n";
$dbh->disconnect();
```

When above program is executed, it will produce the following result:

```
Opened database successfully
Total number of rows deleted : 1
ID = 1
NAME = Paul
ADDRESS = California
SALARY =  25000

ID = 3
NAME = Teddy
ADDRESS = Norway
SALARY =  20000

ID = 4
NAME = Mark
ADDRESS = Rich-Mond
SALARY =  65000
```

CHAPTER

50

SQLite Python

Installation

The SQLite3 can be integrated with Python using sqlite3 module, which was written by Gerhard Haring. It provides an SQL interface compliant with the DB-API 2.0 specification described by PEP 249. You do not need to install this module separately because its being shipped by default along with Python version 2.5.x onwards.

To use sqlite3 module, you must first create a connection object that represents the database and then optionally you can create cursor object, which will help you in executing all the SQL statements.

Python sqlite3 module APIs

Following are important sqlite3 module routines, which can suffice your requirement to work with SQLite database from your Perl program. If you are looking for a more sophisticated application, then you can look into Python sqlite3 module's official documentation.

S.N.	API & Description
1	**sqlite3.connect(database [,timeout ,other optional arguments])** This API opens a connection to the SQLite database file database. You can use ":memory:" to open a database connection to a database that resides in RAM instead of on disk. If database is opened successfully, it returns a connection object. When a database is accessed by multiple connections, and one of the processes modifies the database, the SQLite database is locked until that transaction is committed. The timeout parameter specifies how long the connection should wait for the lock to go away until raising an exception. The default for the timeout parameter is 5.0 (five seconds). If given database name does not exist then this call will create the database. You can specify filename with required path as well if you want to create database anywhere else except in current directory.
2	**connection.cursor([cursorClass])** This routine creates a **cursor** which will be used throughout of your database programming with Python. This method accepts a single optional parameter cursorClass. If supplied, this must be a custom cursor class that extends sqlite3.Cursor.
3	**cursor.execute(sql [, optional parameters])**

	This routine executes an SQL statement. The SQL statement may be parameterized (i. e. placeholders instead of SQL literals). The sqlite3 module supports two kinds of placeholders: question marks and named placeholders (named style). For example:cursor.execute("insert into people values (?, ?)", (who, age))
4	**connection.execute(sql [, optional parameters])** This routine is a shortcut of the above execute method provided by cursor object and it creates an intermediate cursor object by calling the cursor method, then calls the cursor's execute method with the parameters given.
5	**cursor.executemany(sql, seq_of_parameters)** This routine executes an SQL command against all parameter sequences or mappings found in the sequence sql.
6	**connection.executemany(sql[, parameters])** This routine is a shortcut that creates an intermediate cursor object by calling the cursor method, then calls the cursor.s executemany method with the parameters given.
7	**cursor.executescript(sql_script)** This routine executes multiple SQL statements at once provided in the form of script. It issues a COMMIT statement first, then executes the SQL script it gets as a parameter. All the SQL statements should be separated by semi colon (;).
8	**connection.executescript(sql_script)** This routine is a shortcut that creates an intermediate cursor object by calling the cursor method, then calls the cursor's executescript method with the parameters given.
9	**connection.total_changes()** This routine returns the total number of database rows that have been modified, inserted, or deleted since the database connection was opened.
10	**connection.commit()** This method commits the current transaction. If you don.t call this method, anything you did since the last call to commit() is not visible from other database connections.
11	**connection.rollback()** This method rolls back any changes to the database since the last call to commit().
12	**connection.close()** This method closes the database connection. Note that this does not automatically call commit(). If you

	just close your database connection without calling commit() first, your changes will be lost!
13	**cursor.fetchone()** This method fetches the next row of a query result set, returning a single sequence, or None when no more data is available.
14	**cursor.fetchmany([size=cursor.arraysize])** This routine fetches the next set of rows of a query result, returning a list. An empty list is returned when no more rows are available. The method tries to fetch as many rows as indicated by the size parameter.
15	**cursor.fetchall()** This routine fetches all (remaining) rows of a query result, returning a list. An empty list is returned when no rows are available.

Connecting To Database

Following Python code shows how to connect to an existing database. If database does not exist, then it will be created and finally a database object will be returned.

```
#!/usr/bin/python

import sqlite3

conn = sqlite3.connect('test.db')

print "Opened database successfully";
```

Here, you can also supply database name as the special name **:memory:** to create a database in RAM. Now, let's run above program to create our database **test.db** in the current directory. You can change your path as per your requirement. Keep above code in sqlite.py file and execute it as shown below. If database is successfully created, then it will give the following message:

```
$chmod +x sqlite.py
$./sqlite.py
Open database successfully
```

Create a Table

Following Python program will be used to create a table in previously created database:

```
#!/usr/bin/python

import sqlite3

conn = sqlite3.connect('test.db')
print "Opened database successfully";

conn.execute('''CREATE TABLE COMPANY
       (ID INT PRIMARY KEY     NOT NULL,
       NAME           TEXT    NOT NULL,
       AGE            INT     NOT NULL,
```

```
       ADDRESS        CHAR(50),
       SALARY         REAL);''')
print "Table created successfully";

conn.close()
```

When above program is executed, it will create COMPANY table in your **test.db** and it will display the following messages:

```
Opened database successfully
Table created successfully
```

INSERT Operation

Following Python program shows how we can create records in our COMPANY table created in above example:

```
#!/usr/bin/python

import sqlite3

conn = sqlite3.connect('test.db')
print "Opened database successfully";

conn.execute("INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) \
      VALUES (1, 'Paul', 32, 'California', 20000.00 )");

conn.execute("INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) \
      VALUES (2, 'Allen', 25, 'Texas', 15000.00 )");

conn.execute("INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) \
      VALUES (3, 'Teddy', 23, 'Norway', 20000.00 )");

conn.execute("INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) \
      VALUES (4, 'Mark', 25, 'Rich-Mond ', 65000.00 )");

conn.commit()
print "Records created successfully";
conn.close()
```

When above program is executed, it will create given records in COMPANY table and will display the following two lines:

```
Opened database successfully
Records created successfully
```

SELECT Operation

Following Python program shows how we can fetch and display records from our COMPANY table created in above example:

```
#!/usr/bin/python

import sqlite3

conn = sqlite3.connect('test.db')
print "Opened database successfully";

cursor = conn.execute("SELECT id, name, address, salary  from COMPANY")
for row in cursor:
```

```
   print "ID = ", row[0]
   print "NAME = ", row[1]
   print "ADDRESS = ", row[2]
   print "SALARY = ", row[3], "\n"

print "Operation done successfully";
conn.close()
```

When above program is executed, it will produce the following result:

```
Opened database successfully
ID =  1
NAME =  Paul
ADDRESS =  California
SALARY =  20000.0

ID =  2
NAME =  Allen
ADDRESS =  Texas
SALARY =  15000.0

ID =  3
NAME =  Teddy
ADDRESS =  Norway
SALARY =  20000.0

ID =  4
NAME =  Mark
ADDRESS =  Rich-Mond
SALARY =  65000.0

Operation done successfully
```

UPDATE Operation

Following Python code shows how we can use UPDATE statement to update any record and then fetch and display updated records from our COMPANY table:

```
#!/usr/bin/python

import sqlite3

conn = sqlite3.connect('test.db')
print "Opened database successfully";

conn.execute("UPDATE COMPANY set SALARY = 25000.00 where ID=1")
conn.commit
print "Total number of rows updated :", conn.total_changes

cursor = conn.execute("SELECT id, name, address, salary  from COMPANY")
for row in cursor:
   print "ID = ", row[0]
   print "NAME = ", row[1]
   print "ADDRESS = ", row[2]
   print "SALARY = ", row[3], "\n"

print "Operation done successfully";
conn.close()
```

When above program is executed, it will produce the following result:

```
Opened database successfully
Total number of rows updated : 1
ID =  1
NAME =  Paul
ADDRESS =  California
SALARY =  25000.0

ID =  2
NAME =  Allen
ADDRESS =  Texas
SALARY =  15000.0

ID =  3
NAME =  Teddy
ADDRESS =  Norway
SALARY =  20000.0

ID =  4
NAME =  Mark
ADDRESS =  Rich-Mond
SALARY =  65000.0

Operation done successfully
```

DELETE Operation

Following Python code shows how we can use DELETE statement to delete any record and then fetch and display remaining records from our COMPANY table:

```
#!/usr/bin/python

import sqlite3

conn = sqlite3.connect('test.db')
print "Opened database successfully";

conn.execute("DELETE from COMPANY where ID=2;")
conn.commit
print "Total number of rows deleted :", conn.total_changes

cursor = conn.execute("SELECT id, name, address, salary  from COMPANY")
for row in cursor:
   print "ID = ", row[0]
   print "NAME = ", row[1]
   print "ADDRESS = ", row[2]
   print "SALARY = ", row[3], "\n"

print "Operation done successfully";
conn.close()
```

When above program is executed, it will produce the following result:

```
Opened database successfully
Total number of rows deleted : 1
ID =  1
NAME =  Paul
ADDRESS =  California
SALARY =  20000.0
```

```
ID =  3
NAME =  Teddy
ADDRESS =  Norway
SALARY =  20000.0

ID =  4
NAME =  Mark
ADDRESS =  Rich-Mond
SALARY =  65000.0

Operation done successfully
```